Extraction and Integration of Web Query Interfaces

D I S S E R T A T I O N

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. Rer. Nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt-Universität zu Berlin

von
Dipl.-Inf. Thomas Kabisch

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Jan-Hendrik Olbertz

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II:
Prof. Dr. Peter Frensch

Gutachter:
1. Prof. Dr. Ulf Leser, Humboldt-Universität zu Berlin
2. Prof. Dr. Felix Naumann, Hasso-Plattner-Institut Potsdam
3. Prof. Dr. Eberhard Rahm, Universität Leipzig

eingereicht am: 24.01.2011
Tag der mündlichen Prüfung: 13.07.2011

Abstract

Databases on the Web offer large amounts of structured content from various domains. Many popular Web applications, such as comparison shopping systems or search engines, rely on the programmatic access and/or the integration of the content of such Web databases. With the rapid increase of the amount of data available this way, techniques that support a seamless programmatic access of Web databases become increasingly important.

In contrast to relational databases Web databases do not provide interfaces that directly support a programmatic access to the databases content. In contrast, the interfaces focus on human users. Therefore, in comparison with classical database integration, the integration of Web databases requires additional effort to transform Web interfaces into a machine readable representation. The realization of this transformation step is challenging because these interfaces in general do not provide sufficient meta information about their elements, because they lack a common structure, and because logical elements are almost indistinguishable from representational elements.

This thesis focuses on the integration of Web query interfaces. We model the integration process in several steps: First, unknown interfaces have to be classified with respect to their application domain (classification); only then a domain-wise treatment is possible. Second, interfaces must be transformed into a machine readable format (extraction) to allow their automated analysis. Third, as a pre-requisite to integration across databases, pairs of semantically similar elements among multiple interfaces need to be identified (matching). Only if all these tasks have been solved, systems that provide an integrated view to several data sources can be set up.

This thesis presents new algorithms for each of these steps. We developed a novel extraction algorithm that exploits a small set of commonsense design rules to derive a hierarchical schema for query interfaces. In contrast to prior solutions that use mainly flat schema representations, the hierarchical schema better represents the structure of the interfaces, leading to better accuracy of the integration step. Next, we describe a multi-step matching method for query interfaces which builds on the hierarchical schema representation. It uses methods from the theory of bipartite graphs to globally optimize the matching result. As a third contribution, we present a new method for the domain classification problem of unknown interfaces that, for the first time, combines lexical and structural properties of schemas. All our new methods have been evaluated on real-life datasets and perform superior to previous works in their respective fields. Additionally, we present the system VisQI that implements all introduced algorithmic steps and provides a comfortable graphical user interface to support the integration process.

Zusammenfassung

Web Datenbanken enthalten große Mengen von qualitativ hochwertigen strukturierten Inhalten. Viele populäre Anwendungen wie beipielsweise Produktvergleichssysteme oder Suchmaschinen erfordern Methoden für einen programmgestützten Datenbank-Zugriff und die Integration der unterliegenden Inhalte. Durch das starke Wachstum der Datenmenge in Web Datenbanken wird dieses Problem zunehmend wichtiger.

Im Gegensatz zu beispielsweise relationalen Datenbanken unterstützen Web Datenbanken den programmgestützten Zugriff auf ihre Inhalte in der Regel nicht durch geeignete Schnittstellen. Ansätze, die einen automatisierten Zugriff auf Web Datenbanken bereitstellen, können ausschließlich die für menschliche Interaktion konzipierten Schnittstellen nutzen. Daher ist ein zusätzlicher Aufwand erforderlich, um die Web Schnittstellen in eine maschinenlesbare Form zu transformieren. Die Realisierung dieses Schrittes ist komplex, da die Web Schnittstellen keinerlei Metainformation über ihre Elemente bereitstellen und keine einheitliche Struktur aufweisen.

Wir unterscheiden zwischen zwei Arten von Web Schnittstellen: Anfrageschnittstelle (Web Form) und Ergebnisschnittstelle. Die Anfrageschnittstelle ermöglicht dem Nutzer, interaktiv Parameter für eine Datenbankanfrage zu definieren. Die Ergebnisschnittstelle präsentiert die Datenbankrückgaben in einer Web-gerechten Form. Diese Arbeit fokussiert auf die Integration von Anfrageschnittstellen.

Wir identifizieren mehrere Schritte für den Integrationsprozess: Im ersten Schritt werden unbekannte Anfrageschnittstellen auf ihre Anwendungsdomäne hin analysiert, um ein domänenweises Vorgehen in den Folgeschritten zu ermöglichen. Im zweiten Schritt werden die Anfrageschnittstellen in ein maschinenlesbares Format transformiert (Extraktion). Im dritten Schritt werden Paare semantisch gleicher Elemente zwischen den verschiedenen zu integrierenden Anfragesschnittstellen identifiziert (Matching). Diese Schritte bilden die Grundlage, um Systeme, die eine integrierte Sicht auf die verschiedenen Datenquellen bieten, aufsetzen zu können.

Diese Arbeit beschreibt neuartige Lösungen für alle drei der genannten Schritte. Der erste zentrale Beitrag ist ein Exktraktionsalgorithmus, der eine kleine Zahl von Designregeln dazu benutzt, um Schemabäume abzuleiten. Gegenüber früheren Lösungen, welche in der Regel lediglich eine flache Schemarepräsentation anbieten, ist der Schemabaum semantisch reichhaltiger, da er zusätzlich zu den Elementen auch Strukturinformationen abbildet. Der Extraktionsalgorithmus erreicht eine verbesserte Qualität der Element-Extraktion verglichen mit Vergängermethoden. Der zweite Beitrag der Arbeit ist die Entwicklung einer neuen Matching-Methode. Hierbei ermöglicht die Repräsentation der Schnittstellen als Schemabäume eine Verbesserung vorheriger Methoden, indem auch strukturelle Aspekte in den Matching-Algorithmus einfließen. Zusätzlich wird eine globale Optimierung durchgeführt, welche auf der Theorie der bipartiten Graphen aufbaut.

Als dritten Beitrag entwickelt die Arbeit einen Algorithms für eine Klassifikation von Schnittstellen nach Anwendungsdomänen auf Basis der Schemabäume und den abgeleiteten Matches. Zusätzlich wird das System VisQI vorgestellt, welches die entwickelten Algorithmen implementiert und eine komfortable graphische Oberfläche für die Unterstützung des Integrationsprozesses bietet.

Contents

1 Introduction

In March 1989 Tim Berners Lee, at that time an employee of the CERN Institute in Switzerland, proposed a new way to manage information [9]. This was the birth of todays leading infrastructure for the world-wide publishing and presentation of information, the World Wide Web. Since then, the number of Web pages and Web sites has grown exponentially.

Today, the Web has evolved into a data-rich repository containing significant structured content. This content resides mainly in Web databases that are also referred to as the *Deep Web*. Recent surveys estimated millions of such sources [19, 63]. In order to obtain the contents of Web databases, a user has to pose structured queries. These queries are formulated by filling in Web forms. We call a Web page that contains a Web form *query interface*.

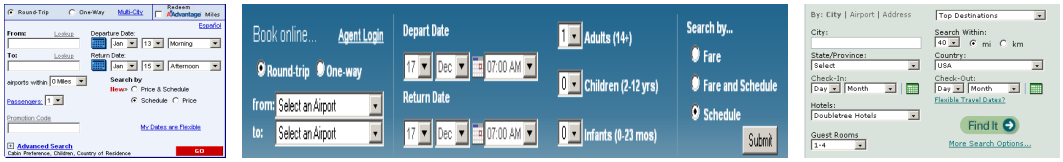
Common examples for Web databases are Web sites for booking hotel rooms or airline tickets. Figures 1.1 (a) and (b) show two typical query interfaces for booking a flight. Both ask for approximately the same types of information (e.g. `departure airport` or `number of passengers`), but their layout is different. Figure 1.1 (c) gives an example for a hotel booking query interface.

Consider the case of booking a flight online. Usually, multiple carriers/airlines provide connections to a particular flight destination. All of them use their own Web site to present only their offerings. A user that aims to find the cheapest or fasted connection among all carriers offering the desired destination has to first probe multiple Web databases, then compare the returned results and finally select the best result. In this scenario, the user *integrates* the returned results originating from multiple Web databases of a single domain. We call this kind of integration *horizontal integration* [53].

Next, consider a combined search for a flight and a hotel at the flight destination. In that case, the user accesses airline and hotel Web sites sequentially. She uses the results of the flight search (e.g. arrival date and time at destination) as the input for the hotel booking database. This is an example of *vertical integration* [53], because it uses multiple sources of different domains.

In the given application scenarios the individual probing of all possible sources is the only way to figure out the best results. With each application domain hosting a large and increasing number of data sources, it is unrealistic to expect the user to manually perform this procedure. Only programmatic solutions that help to integrate Web databases automatically are able to provide solutions for problems like the two examples shown above in large scale and good quality.

Integration may be performed on the data level prior to the online querying process or on interface level online while processing user queries. We discuss both approaches in Chapter 2. Some currently existing integrated systems already provide a unified view



(a) American Airlines (b) Central Mountain Airlines (c) Doubletree Hotels

Figure 1.1: Three example interfaces for Deep Web sources.

over multiple Web databases in selected application domains. These systems mainly follow the data integration paradigm in contrast to interface integration as proposed in this work. Therefore those integration systems can use only that information that has been integrated previously. For example, in the application domain of flight booking integrated services such as the travel service Expedia rely on the data available in the underlying reservation system (for example, Amadeus). Only a few currently existing systems integrate on interface level. In contrast to our generic solution most of these systems adjust the integration layer specifically to a small number of previously identified data sources.

The communication in the Deep Web follows the HTTP request-response model. A user poses a request, the Web server processes the request and sends a response back to the user. The request in a Deep Web scenario is generated from parameters filled in a Web form. The response is dynamically generated from database content that satisfies the parameters of the request. Request and response each correspond to one or multiple Web pages. We call these pages *interfaces* of a Web database. We distinguish between *query interface* and *result interface*. Figure 1.2 illustrates this difference.

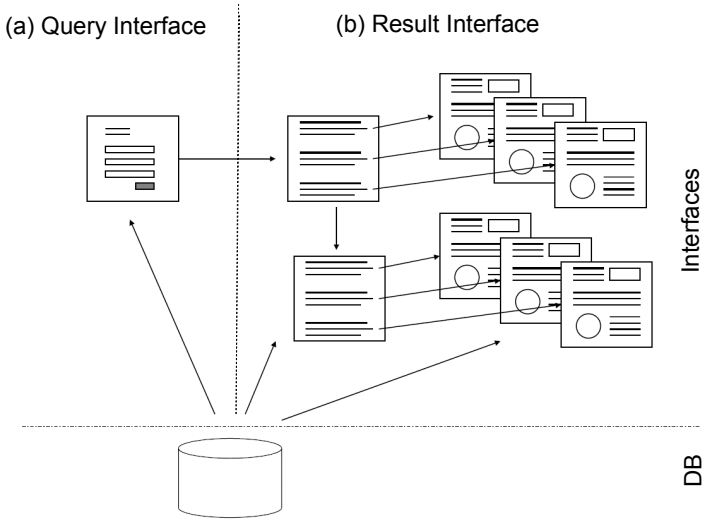


Figure 1.2: Deep Web source and its interfaces.

The interface integration of Web databases is performed in a multi-step process. The integration architecture builds on the two parts of the Web source interface (query interface and result interface, see Figure 1.2) and the requirements of the HTTP communication protocol. Both interface parts require similar steps for their integration. Whereas the contributions of this thesis focus only on query interfaces, we also give an overview about current work on result interface integration in Chapter 2.

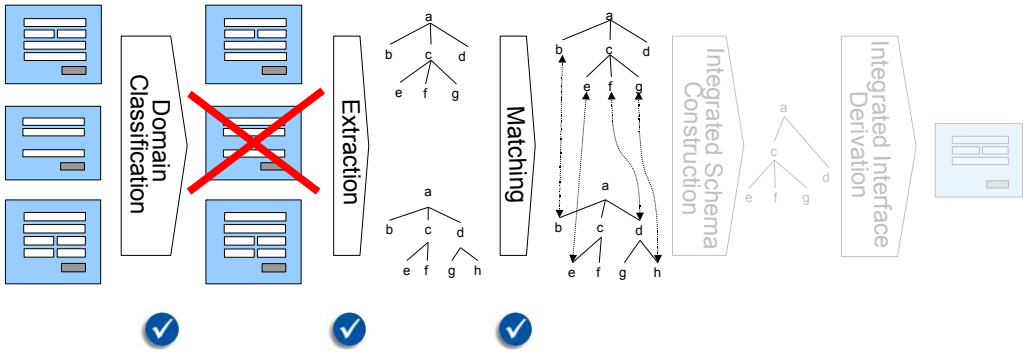


Figure 1.3: Common scenario when integrating Deep Web query interfaces horizontally.

The contributions of this thesis may be applied in multiple integration scenarios. In this work, we concentrate on the specific case of horizontal integration that aims at the establishment of a single unified query interface. The steps of this process are depicted in Figure 1.3. We distinguish the 5 steps *domain classification*, *extraction*, *matching*, *integrated schema construction* and *integrated interface derivation*. In the domain classification step the integration system selects from a number of unknown interfaces those ones to be integrated based on their identified application domain (e.g. car rental). In the extraction step it transforms HTML-based, layout-oriented interfaces into a structured, data-oriented representation, the *interface schema*. In the matching step the integration system identifies conceptually identical elements in the schemas of the interfaces to be integrated. In the integrated schema construction step this mapping information is exploited to construct a unified schema that represents all interfaces to be integrated. Finally, in the integrated schema derivation step this unified schema is transformed back into a query interface that enables to query all integrated Web databases at once.

This work contributes to the first three steps.

1.1 Goals of this Thesis

This work focuses on tools and methods for facilitating the programmatic access to the Deep Web. Its goal is to support developers of Deep Web integration systems by providing a framework to access and integrate databases in the Deep Web. As introduced

before we do not support all steps for the construction of an integrated information system, but, we concentrate on the three central issues of Deep Web interface integration and provide novel solutions for them:

- Extraction of query interfaces
- Matching of query interfaces
- Domain classification of Deep Web sources

In the following, we give a short insight into each of the three goals.

1.1.1 Extraction of Query Interfaces

The success of integration applications in the Deep Web hinges upon two issues: exploiting the structure of Web databases and obtaining their data.

Both issues rely on a good (machine-) understanding of the structure of Web query interfaces, because a query interface provides a glimpse into the structure of the underlying database and is the main means to retrieve data from the database [19].

Deep Web sources do not provide interfaces that allow a programmatic access of their content. In contrast, the content of Deep Web sources is hidden behind a query interface. A system (for instance, an integration system) that aims to use the data of a Deep Web source has to access the content through the query interface. Therefore, an important initial step when focusing on integration in the Deep Web is the transformation of the query interfaces into a machine readable format, a step we call *interface extraction*. Interface extraction may be performed by hand or automatically. The further is cost and time consuming, especially when focusing on integration in Web scale. We concentrate on the automated solution. Therefore, the aim of interface extraction in our understanding is to transform the graphical user interface of a Web form into an *interface schema* automatically.

The process of interface extraction is difficult because Web designers layout their interfaces for human users only. Especially, they do not provide any clues such as meta information that help to automatically derive the underlying database schema. On the one hand, prior works in interface extraction [37, 93] recognize Deep Web interfaces mainly as flat collections of attributes. On the other hand, most integration approaches rely on hierarchical schemas [91]. Therefore, one problem to be addressed in this thesis is to define a uniform way to extract Deep Web interfaces into a hierarchical structure that best represents their semantics.

1.1.2 Matching of Query Interfaces

A key component of integration systems is the recognition of *semantic correspondences* (or mappings) among the data sources. Correspondences connect those schema elements of different interfaces that represent similar semantic concepts. The process of

automatically identifying such correspondences is called *matching*. The correct identification of correspondences is important for many further integration steps, for example the transformation of queries for particular data sources (compare Section 2.1).

The matching of Web query interfaces poses additional challenges compared to schema matching [19, 38]. The reasons are similar to those of the extraction step. Classical databases provide widespread meta information that may be utilized for the support of matching algorithms (compare [73, 1]). For example, structural information as given in the database schema may help to identify dependencies between attributes. Web query interfaces do not provide such clues to support the matching process. Instead, the matching method has to rely on properties of the query interface such as labels or visual arrangements of elements. Although these properties may be exploited to obtain information about the database structure they have been provided for the sole purpose to support the *human* understanding and therefore their correct automated recognition is difficult. The visual clues are reflected by the extracted hierarchical structure of query interfaces. It is the best available approximation of the hidden database schema. Therefore, we focus on a matching method that exploits properties of this hierarchical structure.

1.1.3 Domain Classification of Deep Web Sources

Web query interfaces may be classified in order to their primary application focus. For example, all Web sites that allow to book a flight build the *application domain* `flight booking`. With *domain classification* we mean the automatic analysis of an unknown Web source in order to predict its application domain. Automated domain classification is an important task when integrating Deep Web sources. It has been subject of a number of recent works [62, 5, 35]. On the one hand, the Deep Web contains a huge number of sources. Crawling engines can discover these sources and their interfaces automatically [3]. On the other hand, many techniques of database integration require sources of one single domain (horizontal integration). A cross-domain integration (vertical integration) has only limited application scenarios.

When mapping query interfaces, the previous knowledge of application domains has particular advantages especially during the homonym resolution phase. The number of meanings of a particular term decreases significantly, therefore, the disambiguation accuracy improves.

Our goal is to provide a generic and robust classification approach for Deep Web integration. We show that each domain has their specific *pattern* on a structural level as given in the extracted schema trees. We will use such patterns to classify unknown interfaces to a domain previously learned.

1.2 Example Step-by-Step

Consider again the example of booking a flight to a destination offered by multiple carriers. We are interested in the cheapest flight available for a certain date. As mentioned above, in a manual search we first identify Web sites of suitable carriers. Then we query

all available sites one after each other and collect the concrete flight offers of all sources. Finally we compare the prices and book the cheapest flight.

We now explain how this retrieval could be performed when an integration system is available. Similarly to the manual scenario, we first identify the data sources to be used for the flight retrieval. A Web crawler may walk through the Web by following links and identifying Web pages that contain query interfaces. For each of those interfaces we need to decide whether it belongs to the airline domain or not. We assume that most interfaces of a particular domain follow a specific pattern. We further assume, that the integration system previously learned those pattern for the airline domain from a learning set of interfaces. The system compares each interface found with the learned pattern. If a particular interface matches the pattern of the airline domain, it is assumed to belong to that domain. Consider the two interfaces from Figure 1.1 (a) and (b). Both contain a particular pattern typical for the airline domain which is the sequence `departure date - return date`. Having learned this pattern previously, the system is able to identify novel interfaces as members of the airline domain by matching their elements (such as fields and labels) to the pattern.

Having assigned interfaces to the airline domain we start to integrate them. In the manual scenario, the user next has to "understand" each of the interfaces. For example, she needs to know that the field labeled with `'From:'` in interface (a) stands for the departure airport of the flight.

We may use an extraction algorithm that transforms interfaces into a hierarchical representation (schema tree) to enable an automatic treatment. The potential results for the example interfaces are the schema trees as given on the bottom of Figures 1.4 and 1.5, respectively. In the example, we introduced identifiers for each node in the schema tree for clarity. We distinguish leaf and internal nodes. Leaf nodes in the schema tree represent the fields of the interface. A leaf node may contain the following information: the name of the field (taken from HTML source, denoted in round parentheses), values (predefined value lists or default values, denoted in brackets) and a label (a describing text visible to the user, denoted in plain text). For example, the leaf node G represents the field `From` in Figure 1.4. Interfaces may group fields and form bigger semantic building blocks. Internal nodes in the schema tree represent these groups of fields. Similarly to leaf nodes internal nodes may have an attached label, but never have a name nor values. An example for an internal node in Figure 1.4 is the node H in the schema tree that represents the group of fields labeled `Departure Date`.

Next, the user may start to retrieve the data sources. In the motivating example, she therefore sends *semantically similar* queries to all data sources in order to later compare the results and identify the best (cheapest) flight. More in detail, the user assigns specific query parameters (such as the desired departure date) to the fields of each used query interface. Hereby, she intuitively identifies pairs of elements among the interfaces that denote the same concept such that each source receives the same query. As introduced before, we call such pairs *mappings*.

An example for a mapping on field level is the field labeled with `'From'` in Figure 1.4 (node G in schema tree) and the field denoted with `'from:'` in Figure 1.5 (node f in schema tree). An example for a mapping of groups is `'Departure Date:'` (node H)

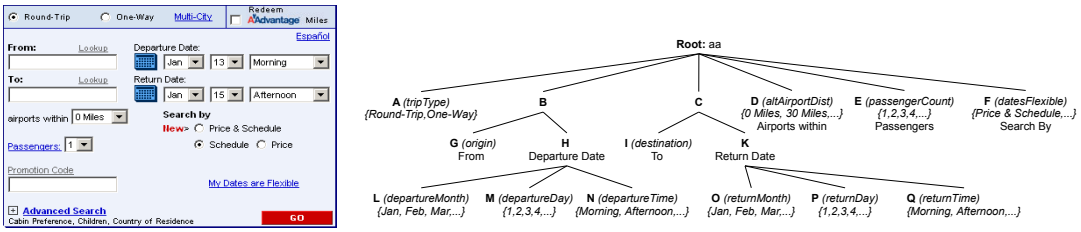


Figure 1.4: Interface of American Airlines together with its schema tree.

in the first interface (Figure 1.4) and `'Depart Date'` (node h) in the second interface (Figure 1.5).

The integration system identifies mappings in the automated integration scenario by using a matching algorithm. It computes a similarity score between elements. For example, the algorithm might compute the matching score for the above mentioned mapping (G, f) to be 1 because both fields share the same label `'From'`.

Such a matching algorithm may compare also other properties such as values. Nevertheless, a field-based approach does not lead to correct mappings in all cases, although it seems to be a natural way to perform the mapping between interface elements bottom up starting from single fields. For example, a field-based matching algorithm is likely to construct a wrong a mapping between the field `departure month` (node L in Figure 1.4) and the field `return month` (node s in Figure 1.5) because both nodes share similar value sets and parts of their labels.

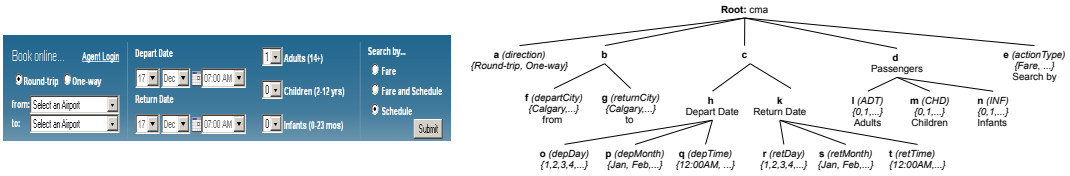


Figure 1.5: Interface of Central Mountain Airlines together with its schema tree.

In General, such misleading mappings may occur due to similar value sets and/or missing field labels. Additionally, field labels may be homonymous. Therefore, we need to consider additional properties of the schema trees to match their elements. For example, we may determine the semantics of the fields by considering the groups and their mappings. In our motivating example, one mapping between groups of fields is (H, h). Finally, we may map a field of one query interface to a group of another query interface. In our motivating example, we map the field denoted with `'Passengers'` (node E) in Figure 1.4 to the group of fields (`Adults, Children, Infants`) in Figure 1.5 (node d).

The knowledge about mappings enables the integration system to construct semantically correct queries for each data source to be integrated. For example, we may trans-

form the HTTP query fragment '`...?origin="Berlin"&destination="Chicago"...`' that is valid for the query interface in Figure 1.4 into '`...?departCity="Berlin" &returnCity="Chicago"...`' that corresponds to the query interface in Figure 1.5. This transformation utilizes the mappings (G, f) and (I, g) that map the fields for the origin and destination of the flight retrieved. However, such *query planning step* is out of the scope of this work. We give only a short overview in Section 2.1, for further details please refer to Leser and Naumann [53].

Having submitted the queries and returned the results from the sources, the user compares the results and selects the best one. Although the steps of the *result integration* are also not in the focus of this work, in an automated setting the mappings may support an integration system by identifying conceptually identical entities among the returned results. We give some information about the related work in this area in Chapter 2, Section 2.3.

1.3 Contributions and Prior Work

We now characterize the contributions of this thesis. Parts of this work base on prior cooperative work with other authors. For these parts, we assign the specific contributions to their authors.

1.3.1 Contributions of this Thesis

The four main achievements of this thesis are:

A novel Algorithm for Query Interface Extraction The schema tree of a query interface exploits the graphical structure of the query interface and thus best represents the intended meaning of it. We construct the schema tree in a three step process. Therein, we utilize visual clues of query interfaces such as order and alignment of its elements. First, we identify the labels and the order of the fields. This information is stored in an initial *tree of fields*. Then, we investigate relationships among the labels of the interface. We store this information in a *tree of text tokens*. Both trees reflect different properties of query interfaces. We finally construct the schema tree from the two initial trees. We transform the query interface extraction problem into an tree integration problem [24]. The accuracy of the extraction approach averages 90%. The algorithm is the first query interface extraction approach that represents interfaces as semantically rich trees. Also, it outperforms prior approaches for field extraction by about 6% on accuracy.

A novel Query Interface Matching Approach Driven by the human way of recognition we focus on *structural mappings* that we characterize by the following properties: (1) local features of single elements such as label, name or values, (2) the ancestor and sibling relationships among nodes in the schema tree, and (3) the order of the elements in the schema tree. These properties are combined into an approach to map interface elements on different abstraction levels automatically. We use an

optimization technique from graph theory [12] to find a global optimal matching between two query interfaces. We tested our matching method on interfaces of 7 application domains. The average accuracy of the method over all domains tested is about 80%. In some domains it reaches more than 90%. We also implemented a greedy method as suggested by [91] to compare our algorithm with their results. We show, that on the dataset we tested our approach outperforms the greedy approach by approximately 4%.

A Domain Classification Method for Deep Web Sources We use the set of mappings among interfaces of a learning set to derive *interface domain patterns*. We cluster all elements in the learning set by their concepts. These clusters help to derive patterns. The domain classification algorithm first extracts the unknown interface into its schema tree representation. It matches the nodes of the schema tree to the pattern of each known domain. It assigns the interface to the domain containing the best matching pattern. Additionally, we developed a neighbor-based classification method that computes similarities between interfaces and utilizes these similarities to predict the application domain of an unknown interface. The classification method was tested on interfaces of 7 application domains, the average f-measure of the method is about 80%. A precision of 100% for the classification can easily be adjusted retaining a considerable recall.

The Integration Framework VisQI We implemented VisQI (*Vis*ual *Q*uery Interface *I*ntegration System), a system that enables the user to perform the three steps of query interface integration as previously introduced. The system provides a graphical user interface that allows in-depth validation of all algorithmic steps. VisQI has a sophisticated testing component which is accompanied by large gold standard sets for both query interface extraction and matching. An intuitive visualization component supports multiple steps when constructing integration systems. Its open architecture allows to exchange components and reuse the overall system. Therefore, VisQI is a framework to support multiple purposes when performing Deep Web integration.

1.3.2 Assignment of Contributions to Authors

The scientific contributions in Chapter 3 and Chapter 6 of this thesis have been founded and published as a result of collaborative work during multiple research stays at the University of Illinois at Chicago in 2007 and 2008. The resulting publications "A Hierarchical Approach to Model Web Query Interfaces for Web Source Integration" [24] and "Deep Web Integration with VisQI" [43] have multiple authors. Also, parts of the PhD thesis of Eduard Dragut [22] overlap with this thesis because of the collaboration.

This subsection identifies the particular parts of this thesis that originate in the research collaboration and assigns the corresponding contributions to their authors.

Chapter 3: Query Interface Extraction Chapter 3 describes a novel query interface extraction algorithm. All sections of the chapter contain contributions of [24]. Sec-

tion 3.1 defines *Commonsense Design Rules*. These design rules have been identified in a joint work of Eduard Dragut and Thomas Kabisch and therefore cannot be assigned to a single author. In Section 3.2 *The Extraction Algorithm*, we can identify two main algorithmic contributions: (i) the usage of a visual approach for the split-up of the content of Web pages, and, (ii) the concept to use a semantic scope of textual elements for the construction of schema trees. Section 3.2.1 *Token Extraction* and Section 3.2.2 *The Tree of Fields* describe the first aspect and can be assigned mainly to Thomas Kabisch whereas Section 3.2.3 *The Tree of Text Tokens* and Section 3.2.4 *Integration* focus on the second aspect and have been contributed mainly by Eduard Dragut. Section 3.3 *Experimental Evaluation* contains the experimental writeup that has been contributed by Thomas Kabisch.

Additionally to the aspects documented in Chapter 3, Eduard Dragut defined gold standard data sets. Thomas Kabisch implemented a prototype and presented [24] on the conference VLDB 2009 in Lyon.

Chapter 6: Visual Query Interface Integration System VisQI This chapter focuses on the system VisQI that allows to investigate and visualize several aspects of query interface integration. Its main contributions have been published in [43]. Section 6.1 *System Components* describes the architecture and the components of the system. This part and the implementation of VisQI has been contributed by Thomas Kabisch. Section 6.2 *Usage of VisQI* introduces usage scenarios of the system. Eduard Dragut defined the majority of these scenarios. Also, he supported the development with ideas for the visualization and provided Gold Standard data sets.

Additionally to the contributions documented in Chapter 6, Thomas Kabisch provided experimental studies and demo scenarios of VisQI. He also presented the system on the conference VLDB 2010 in Singapore.

1.4 Structure of this Thesis

This thesis is structured as follows. Chapter 2 describes the fundamentals of data and information integration. It describes the concept of information integration with a special focus to virtual integration approaches in Section 2.1. We introduce relevant concepts and technologies of the Web in Section 2.2. Section 2.3 brings both aspects together by discussing information integration in the focus of the Web.

Chapter 3 focuses on the first contribution and presents a novel approach to query interface extraction. We first outline a model for query interfaces and introduce fundamental commonsense rules for their design in Section 3.1. We define each step of the extraction algorithm in Section 3.2. The evaluation of the algorithm is described in Section 3.3. We finally discuss work related to the extraction approach in Section 3.4.

Chapter 4 presents our second contribution, a novel query interface matching algorithm. Section 4.1 describes the overall matching framework whereas the following sections concentrate on its details. Section 4.2 provides information about the preprocessing

step of domain dictionary construction. Section 4.3 defines similarity measures that help to compute element mapping candidates. Section 4.4 describes two methods to achieve a globally optimal matching between two query interfaces from their element mapping candidates and outlines the overall matching algorithm for a set of query interfaces. Section 4.5 gives insight in our experiments and lays out the results of the matching evaluation. Finally, Section 4.6 discusses the related work and limits of the matching approach.

Chapter 5 describes the third contribution, a novel domain classification approach for Web query interfaces. We introduce two different methods for domain classification, pattern based classification (Section 5.1) and neighbor based classification (Section 5.2). Both methods are evaluated separately in Section 5.3. In Section 5.4 we compare both methods and identify advantages and disadvantages for each of them. Finally, we discuss related work in Section 5.5.

Chapter 6 focuses on the system VisQI that implements a framework for Deep Web integration and provides a graphical user interface for all contributions introduced before. We describe its principal architecture in Section 6.1. Possible usage scenarios are outlined in Section 6.2.

Finally, Chapter 7 concludes this thesis. It gives a summary (Section 7.1) and an outlook to potential future development directions (Section 7.2).

2 Fundamentals of Deep Web Integration

This chapter introduces the two main fundamentals of this thesis, *information integration* on the one hand and the *Deep Web* and its technical roots in the World Wide Web on the other hand. Section 2.1 introduces main concepts of information integration. It focuses on those methods that can be adopted to Web integration. Section 2.2 gives an insight into particular issues of the Web, introduces the concept of the *Deep Web* and discusses access methods for it. Finally, Section 2.3 brings both aspects together and describes methodology and selected approaches of Deep Web integration.

2.1 Information Integration

Information integration has become a central issue of research community and computer science industry during the last years [32]. The reason is, on the one hand, the increasing number of data sources and applications. On the other hand, rapidly growing communication infrastructures allow to build networks between former isolated solutions. Together with this technical developments, new application scenarios demand new capabilities when integrating software and its data and information. Information integration is crucial for both, horizontal as well as vertical scenarios.

A central challenge of information integration is the heterogeneity of data representation and storage formats due to the independent development of the data sources and applications. Information integration provides solutions to overcome these heterogeneities and to allow a unified access to the underlying content. We give only a short overview to some issues that are relevant for this work. Leser and Naumann [53] provide a deeper insight to information integration.

We first classify integration systems into *physical information integration* and *virtual information integration* by the way the data is stored. Next, we show architectures for virtual integration systems. We describe, how systems of different architecture styles influence the implementation of the central tasks of the integration. We introduce two central aspects of a virtual integration system: First, when defining an integration system, the designer needs to understand the schemas of the sources and to find their semantically corresponding elements. We denote all those tasks by *schema management*. In contrast, *query processing* subsumes all steps the system performs when answering a user query.

This thesis mainly contributes to schema management, but the results obtained by the matching algorithm (Chapter 4) may also be used during query processing. Beside of this, schema management and query processing are separated in the classical understanding. The further is an offline process, whereas the later is performed for each user query online. There are newer approaches that shift parts of the schema management

into the query processing step when integrating sources on-the-fly. An example of such a system is MetaQuerier [38]. Therefore, we introduce both, schema management as well as query processing aspects of virtual integration systems in this section.

Physical Integration

Physical integration systems are centralized. They materialize a copy of the data from all underlying data sources in a central repository that follows a *global schema*. The global schema has been derived from the *local schemas* of the sources. Physical integration systems execute user queries only against the replicated data. There is no interaction that involves the underlying sources when querying physical integration systems.

Physical integration is popular for *online analytical processing* (OLAP) applications where a long term archiving of data and offline analysis capabilities are targeted. The most popular kind of physical integration applications are *data warehouses* [42].

The classic example of a data warehouse in an enterprise is a management information system. It collects all business transaction data about sales, products, etc. and aggregates these data such that managers get a good picture about long term developments of the enterprise when analyzing the data. Such analytical queries are rather complex.

Because OLAP systems work atop the underlying sources they preserve the autonomy and independence of them. This is especially important when the integration system also manipulates the data. Following the physical integration approach, such manipulations are possible without permission of the underlying sources. Another advantage of physical integration is the good query performance when processing complex queries. The centralized data storage avoids communication overhead and possible network delays.

A disadvantage of a physical integration is the potential risk of data inconsistency between the source data and the replicated data. The integration system needs to keep track with all data changes of the underlying sources. Therefore, physical integration is not useful for applications with highly dynamic contents.

Virtual Integration

In contrast to physical integration, virtual integration avoids persistent replication of data. It leaves all data at the sources and enables a uniform view of them. A global schema is provided similarly to a physical integration system, but, in virtual integration systems, this schema is not instantiated and therefore it is called to be *virtual*. Virtual integration systems integrate the data of the sources online during query processing. The main advantage of virtual integration is the consistency of all data because the integration system does not persist replica. All queries are executed against the original source data. Therefore, the data retrieved by the integration system always represents the current state of all underlying sources.

The higher degree of source independency is another advantage of virtual integration compared to physical integration. The sources (and thus also their owners) may not even know about the existence of the integration system. All efforts are initiated from the integration system itself (pull process). In contrast, in a physical integration system,

the integration process typically is initiated from the sources (push process). Therefore, we can deploy a virtual integration system without permission of the underlying sources. This feature makes virtual integration interesting for many applications where underlying sources (and data owners) are unknown and/or not willing to cooperate. Examples of such sources can be found mainly in competitive environments. Our case study MiWeb [15] implements a virtual integration system in a Web context.

The main disadvantages of virtual integration are the complex querying process and the delays during a query execution. Queries against the integrated view have to be transformed into separate sources queries, forwarded to the data sources and executed at the sources. The results have to be retrieved from sources and re-transformed into the integrated view.

Another crucial point of virtual integration systems, especially in the Web context, is the limited interface between the data sources and the integration system. The system relies on the existing interfaces of the underlying sources. For example, in a Web context there is no interface available that is intended for integration purposes, but virtual integration systems may reuse the interfaces created for human users of the Web sources. Generally, the integration system has only a limited access to the capabilities of the underlying sources.

Source autonomy also prevents the integration system to update data in those sources. Therefore, most virtual integration systems provide a read-only access to the data. This complicates the integration, because data normalization or cleaning such as performed in physical integration systems is not possible when integrating virtually.

2.1.1 Architectures of Virtual Integration Systems

There are many possible architectures for virtual integration systems. Historically, integration efforts considered only (relational) database sources [32]. The database community introduced the integration architectures *distributed database systems*, *multi database systems* and *federated database systems*. The main distinction properties among them are, (1) the degree of source autonomy and, (2) the tightness of coupling sources.

In a distributed database system [70] the sources are tightly coupled, centrally controlled and have no autonomy. In contrast, multi database systems [58] consist of loosely coupled autonomous databases. A multi database language allows to query the databases. Examples of multi database languages are, among others, SchemaSQL [49] and MSQL [57]. A multi database system holds no integrated (global) schema. Federated database systems provide an integrated schema, but in contrast to distributed database systems, the sources preserve their autonomy. The integrated schema provides database-like access to the integrated view.

Federated information systems [78] (Figure 2.1) generalize federated database systems to arbitrary sources. A federated information system distinguishes between a coordinating federation layer and a foundation layer containing the data sources.

Wiederhold [83] identified two types of components in a federated information system, a *mediator* component and a *wrapper* component. He derived the *mediator-based information system architecture (MBIS)* from the more general architecture of federated

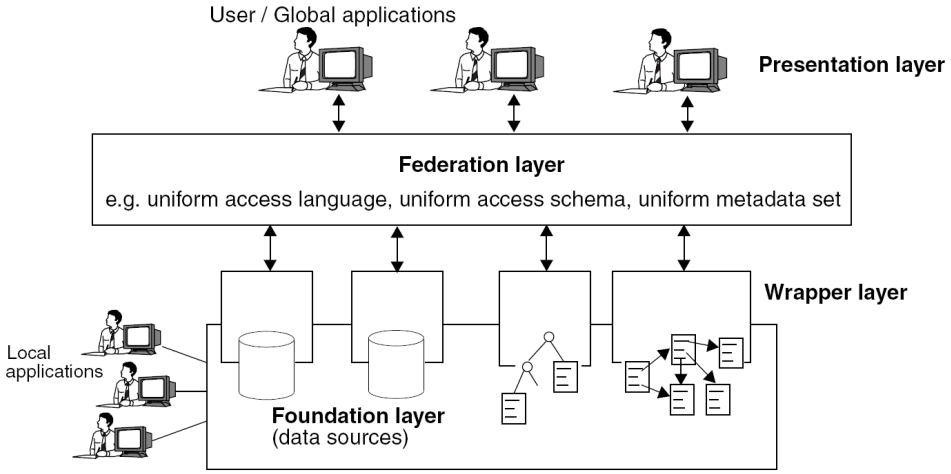


Figure 2.1: Architecture of a federated information system according to [14]

information systems.

In a mediator-based information system the mediator is responsible for all processes necessary for the integration such as the transformation of queries and the integration of the results. When accessing a data source, the mediator communicates with the wrapper of the particular source. The main task of a wrapper is to transform queries and results between the representation of the global mediator schema and those of the particular source wrapped.

Sources that use a different schema from that of the mediator are integrated by defining *correspondences* between the elements of the schemas. The explicit specification of correspondences allows the integration and change of sources during the runtime of the system – a prerequisite for the integration of autonomous Web sources. The main difficulty is the definition of the mediator schema as it has to cover all aspects of the whole system. Therefore, metadata standards are used as mediator schemas in heterogeneous application scenarios like the Web [46], for example the Dublin Core [27] or domain-specific ones. Example implementations of mediator based information systems are TSIMMIS [56], Garlic [76] and MiWeb [15]. TSIMMIS introduced the object oriented data model OEM. Garlic focuses on the query optimization. MiWeb utilizes RDF and the metadata standard LOM [50] to integrate Web sources in a learning context.

In the next subsections, we focus on federated information systems. We discuss two main issues of them, *schema management* and *query processing*. From the information viewpoint, schema management is the most important task during the creation of a federated information system. In contrast, query processing subsumes all tasks of the integration system while answering user queries.

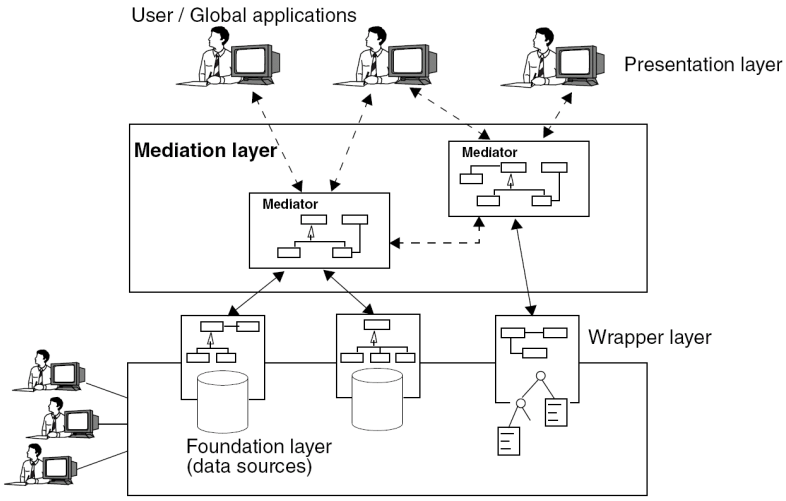


Figure 2.2: Architecture of a mediator-based information system according to [14]

2.1.2 Schema Management

Schema Management subsumes techniques to overcome structural and semantical heterogeneities on the level of schemas [6]. A central point in schema management is the recognition of pairs of conceptually identical elements (*mappings*) among the source schemas. This process is called *schema matching*. More general, we derive *correspondences* from the mappings that describe the semantic relations among elements of the schemas of different sources. The integration system later uses the correspondences for the query processing. Later in the text we will not differentiate between mapping and correspondence and use them interchangeably.

Schema Integration

Schema integration techniques merge multiple source schemas and derive a single integrated schema. Batini et. al. [6] identifies four steps of schema integration: preintegration, comparison of schemas, conforming of schemas and merging/restructuring. A later approach [79] summarizes the first two steps to *investigation* and the last two steps to *integration*. Generally, a schema integration technique first investigates the schemas, and identifies the mappings between conceptually similar objects. Consecutive integration steps use these mappings to unify and merge the schemas.

Two popular approaches that focus on schema integration are correspondence-based methods (for example [79]) and the Generic Integration Model [77].

Schema Matching

In contrast to schema integration, *Schema matching* preserves the source schemas and focuses on the identification of correspondences [17]. Therefore, it encompasses meth-

ods that automatically derive mappings between schemas or elements of schemas. A schema matcher analyzes schemas with respect to their structure, integrity constraints and example data. It derives suggestions for correspondences that domain experts verify prior to the schema integration. A correspondence-driven schema matching tool uses sets of attributes of the input schemas for the matching. When considering only two input schemas, it compares each attribute of the first schema with each attribute of the second schema. The matching tool computes a similarity score between each two attributes based on several heuristics. Examples for heuristics are string edit distance-based measures to compare labels or names or a comparison of attribute value sets. If the similarity score meets a certain threshold, the matching algorithm suggests a correspondence/mapping between the two attributes.

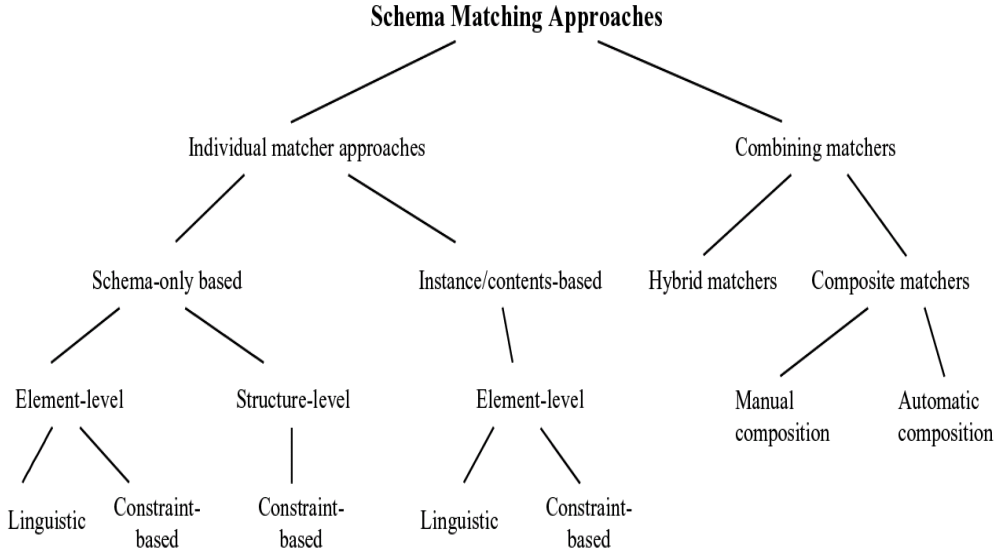


Figure 2.3: Classification of schema matching approaches according to [73].

Figure 2.3 gives an overview about different classes of schema matching methods following Rahm and Bernstein [73]. They first classify the methods due to their general structure and therefore distinguish methods using only one individual matcher and methods combing multiple matchers. Individual matchers rely on a single similarity measure whereas combining matchers bring multiple matching methods together. Combining matchers may either be hybrid - they combine the final measure from multiple measures and provide a single result, or, alternatively they deliver multiple separate results. The combination of these results is independent from the matcher and performed afterwards.

Among the individual matchers, Rahm and Bernstein differentiate between schema-only based methods and instance/contents-based methods. The further uses only the schema information (attributes and their relations) whereas the later investigates instances for finding mappings. The finer differentiation further distinguishes the data and metadata used for matching. Schema based linguistic methods investigate only

names and descriptions of schema elements. Linguistic methods on instance level look for values of attributes. Constraint-based methods on schema level investigate data types or key-relations whereas, on instance level, they look for value patterns.

A newer approach for schema matching not covered by Figure 2.3 is the duplicate-based method [11]. It assumes that the entity sets of the schemas to be integrated overlap. The method looks for duplicates and suggests correspondences between those attributes whose value sets contain duplicates.

2.1.3 Query Processing

During query processing a user queries the global (integrated) schema. She therefore issues a *global query*. Because all data is stored at the sources, the system transforms global queries into multiple *local queries* that suit to the capabilities of the data sources.

In this subsection we summarize processes that a federated information system performs to answer a user query. First, we introduce steps for query processing, second, we discuss the implementation of them in a mediator-based architecture.

Steps of Query Processing

Leser [52] separates query processing into the five steps *query planning*, *query translation*, *query optimization*, *query execution* and *result integration*.

Query Planning: The query planning component splits the global query into multiple local queries that each fits to one or more local data sources. It uses meta information that describes the sources capabilities (*query capabilities*). The result of the query planning step is a set of *query plans*. Each plan consists of a number of queries computing a partial result. The combined result of those queries is the result of the global query.

Query Translation: During query translation the partial queries of all query plans are transformed from the global query language to the local query languages of the sources. The result of this step is a set of locally executable queries.

Query Optimization: The optimizer decides along several assumptions how to execute a query plan in an optimal way. Optimizing criteria are mainly the order of execution and the assignment of predicates to sources.

Query Execution: During query execution the local sources process the transformed parts of the query that have been assigned to them. The integration component watches over the execution. In some cases, a re-planning may be useful.

Result Integration: The integration component aggregates results of the execution of all plans and transforms it into a global representation. One issue of result integration is the identification of duplicates.

Implementation in a Mediator-based Architecture

In mediator-based information systems, there is no single point of execution during query processing. Instead, the above described steps of query processing are distributed between the central mediator, the sources wrapper components and the source systems.

The mediator is solely responsible only for query planning and result integration. The other steps query transformation, query optimization and query execution are shared between the mediator and the wrapper components. The degree of autonomy of the wrappers differs among the implemented systems, for examples see [55, 56, 15].

Generally, wrappers transform queries between the querying language of the mediator and the source specific query language. This approach allows to plug-in sources easily without modifying the mediator component.

We implemented the system MiWeb [15], a case study for the application domain of learning contents that implements the MBIS architecture. MiWeb uses three component types, a mediator for the query planning and result integration, source specific wrappers for transformations between different representation languages and a mapper component for bridging the semantic heterogeneities between the used vocabularies of sources/wrappers and the mediator vocabulary. The vocabulary is standardized using metadata standards.

During the case study MiWeb we performed extraction and mapping of Web query interfaces mainly by hand, which was laborious and error prone. We realized the lack of tool support for this tasks. This observation lead to the central idea of this thesis.

2.2 The Deep Web

We may classify Web sites by the type of the content on their pages. Roughly, there are *unstructured* and *structured* pages. The first class mainly contains static HTML pages whereas the second class is driven by databases that provide an interface to the Web. The variation in the degree of structure is caused by different creation processes. Human users design unstructured pages similarly to documents. They do not follow strict design conventions. In contrast, structured pages are created from the content of Web databases using predefined *design templates*. All pages generated by a single database therefore follow the same structure as encoded in the database schema and the used template.

Search engines such as Google index mainly the unstructured portion of the Web. They access the Web pages and their contents by automatically following hyperlinks. We call this automatic analysis of structure and content of the Web *crawling*. Portions of the Web that tools can access automatically through crawling we refer to as the *Surface Web*.

In contrast, search engines cannot index the contents of most Web databases, because Web databases create their pages dynamically in response to specific submitted user queries. The contents of Web databases are *hidden* for crawlers. Consequently, we call the portion of the Web containing Web databases *Hidden Web* or *Deep Web* [31]. We refer to Web databases also as *Deep Web sources*.

In between the both extremes in the degree of structuring, structured content also emerges from annotation schemes such as that of Flickr where users can tag objects [63]. Although structured, this kind of Web information is highly heterogeneous and does not follow a certain schema such as Web pages that have been generated from database content. We concentrate on classical Web databases.

Many investigations that focus on the estimation of the size of the Deep Web underline its relevance. The problem of estimating the size of the Deep Web has been tackled by recent research [63, 19, 8]. There are two major questions. (1) How many distinct Deep Web entry points (forms) exist? (2) How big is the content behind the entry points? We know no recent work that answers question (1) in a satisfying way, but a white paper of the year 2000 [8] estimated the number of hidden Web pages to about 550 times larger than that of the Surface Web. Different approaches [8, 19, 63] investigated question (2). Bergman et. al. [8] first estimated the number of Deep Web databases in the year 2000 by using overlap analysis between pairs of search engines. Their result numbers range from 43,000 to 96,000 pages. Chang et. al. [19] in 2004 generated a random IP sample of the size of 100,000 IP addresses. For this sample they estimated a number of 281 Web servers. The sample contained 129 query interfaces within a depth of three in the page-structure. It hosted 34 Web databases. Scaling the sample to the size of the entire Web, this study estimated 450,000 Web databases that were accessible through 1.3 Million query interfaces. Another survey of 2007 [63] used a different estimation approach. It sampled 25 million Web pages from the Google index. In contrast to [19], this study figured out that most of the pages in the sample set (23.1 million) contained at least one Web query form. The majority of these forms did not lead to Deep Web content. Instead, the HTML form-technology on this pages was used mainly for a simple page search. The application of several filtering rules reduced the number of distinct Web forms to 2.5% of the number of sample pages. Scaling to the whole Google index finally lead to an estimation of around 25 million Deep Web sources. Although the measuring methods and results differ, we note that the amount of Deep Web content has grown rapidly during the last years.

In the remaining part of this section, we discuss different aspects of the Web and its technologies. We put the focus on the *Deep Web* that emphasizes the hidden content of the Web. After defining the central concepts of the Deep Web we characterize fundamental and newer Web technologies that may help to reduce heterogeneity problems when integrating Deep Web sources. We discuss limits of their application in real world scenarios and underline the need of alternative integration solutions that do not require adjustments on Web server side.

2.2.1 Definitions

We first define central concepts of the Deep Web that are important for this work.

Definition 2.1 (Deep Web Source) *A Deep Web Source is a database driven Web site that cannot be indexed by search engines. It contains at least one Web form that allows the user to pose queries to the database. A Deep Web Source generates result*

pages dynamically for each user query from database content.

Deep Web sources generally provide two interfaces, the query interface and the result interface. The query interface represents the only access point of a Deep Web source. In contrast, the result interface expresses the results of a particular user defined query. We define both interfaces as follows:

Definition 2.2 (Query Interface) *A Web query interface is an access point to a Deep Web source. It is a Web form that enables users to pose queries.*

Definition 2.3 (Result Interface) *The result interface of a Deep Web source is the set of dynamically generated pages that represents the results of a query issued via the query interface. Result pages are generated from Web templates and thus they follow predefined structural patterns.*

A query interface contains of a set of query interface elements:

Definition 2.4 (Query Interface Element) *A Query Interface Element is a visible entity on a query interface that refers to a particular concept of the application domain. We distinguish between simple elements that correspond to a single HTML field and complex elements that resemble a group of fields.*

Both, query interface as well as result interface reflect aspects of the underlying database schema. Further, there is a semantic overlap between query interface and result interface. The elements of an interfaces (such as fields etc.) are ordered, visually grouped and aligned. Together, they form the *interface schema*. Although, we concentrate only on query interfaces in the following, many ideas apply also to result interfaces.

We define an interface schema as follows:

Definition 2.5 (Interface Schema/Schema Tree) *The schema S of a query interface, also called interface schema, is an ordered tree consisting of a set of nodes N and a set of edges E. N represents the elements of the query interface. Therein, leaf nodes correspond to fields and internal nodes correspond to groups of fields. E describes structural relations between elements.*

2.2.2 Technologies of the Web

This subsection briefly describes relevant technologies of the Web. First, we give some overview about the Web fundamentals HTML and HTTP. Second, we introduce XML, the Semantic Web and Web Services - three newer developments in the Web context that

may help to ease the programmatic access to the Web. Third and finally, we discuss real world problems that often prevent the adaption of such techniques at design time. We underline the demand for alternative design-independent ways to a programmatic Web access.

HTML Language

The *HyperText Markup Language* (HTML) is the standard for publishing content in the World Wide Web. HTML is a text *markup* language [86]. Text markup languages are sets of annotations to a text that describe, for instance, how the text is structured, laid out or formatted. In HTML, these annotations are called *tags*. A major reason for the popularity of the Web is the concept of *hyperlinks*. Authors annotate their texts by hyperlinks that point to other related documents. Text documents annotated with such hyperlinks are called *hypertexts*.

HTML is based on the Standard Generalized Markup Language (SGML). SGML has a much richer set of annotations than HTML and was developed for scientific purposes[1]. Besides format tags, HTML documents may contain other information, such as meta-information or *Web forms*. Web forms allow the user to fill in fields with specific parameters. In many cases, Web forms connect to a hidden database. A server-side application transforms the parameters into a database request. Therefore, Web forms enable user interaction and allow to dynamically publish the content of structured databases on the Web.

Web browsers display HTML-formatted Web pages (short Web pages). These tools contain an *HTML parser* and a *rendering engine*. The former interprets the language structure of an HTML document, the later enables a graphical representation of it.

In contrast to classical parsers, HTML browsers use a relaxed language model that allows to interpret also syntactically incorrect HTML. This feature is one of the reasons for the success of HTML. On the one hand, it allows unexperienced authors to publish HTML documents. On the other hand, this technique complicates the automated analysis of HTML documents because a parser may run into ambiguities that are introduced by errors.

HTTP Protocol

The Hypertext Transfer Protocol HTTP is a stateless communication protocol that was created to enable the transfer of hypertext documents in the Web. In the OSI/ISO model for communication[2] it is allocated in the application layer. HTTP communication is message-oriented. We distinguish two types of messages: *request* and *response*. In a regular communication scenario a Web client (usually a browser) sends HTTP-requests to a Web server. A request contains a requested address (URL) and optionally, parameters. The server responds by an HTTP-response that usually contains an HTML-document. Each HTTP-message consists of a header and a body. The header contains

[1] for more information about SGML see http://www.w3.org/MarkUp/SGML/
[2] see http://standards.iso.org/

meta-information about the body. The body contains the information itself, i.e. the HTML-document.

XML

The Extensible Markup Language XML [89] was first introduced in 1997 to provide a method that eases the exchange of structured content. It is based on a hierarchical data model. XML is characterized as *semistructured*, *self-describing* language and uses *tags* in a similar syntax as HTML. The difference to HTML is that XML-tags are user-defined. Whereas HTML-tags mostly describe formating instructions, XML-tags in general do not carry any fixed meaning. Therefore, programs cannot uniformly interpret them. XML is made to allow the definition of user-specific languages by following XML conventions.

We may validate XML-documents in a two step process. First, XML-documents that follow fundamental conventions of the hierarchical data model of XML are called *conform* to XML. Second, the author of a specific XML language may define additional rules that describe valid application cases in a specific scenario. A schema description language (XML Schema) allows multiple ways to constrain XML-documents syntactically, for instance by defining a set of valid tags. A document that follows all defined rules of its schema definition is called *valid*.

Although XML can be utilized for a wide range of purposes, the main usage is to create self-describing data-holding documents. Currently, XML is very successful when exchanging data between heterogeneous information systems (especially in B2B applications). Representing data in XML leads to a reduction of syntactical heterogeneities.

Semantic Web

In the classical World Wide Web, information provides no hint about its meaning. Therefore, it is difficult to unveil the semantics of Web sites automatically. The Semantic Web aims at the automated interpretation of contents in the Web to enable interoperable Web information systems. People adopting the Semantic Web idea understand the Web as a huge distributed information system where services exist that allow an interconnection of arbitrary contents. Tim Berners Lee laid out the fundamentals of the Semantic Web in 2001 [10].

The central idea of the Semantic Web is called *semantic tagging*. Semantic tagging describes the enrichment of information items by additional meta information that defines the meaning of the information. Different communities have a different understanding of particular terms and concepts. This is a central problem when exchanging information. Therefore, people, who intent to exchange information need to ensure a *common understanding* of their information. Semantic Tagging is a technique the eases the establishment of a common understanding by providing publicly available vocabularies. We characterize semantic tagging by two steps: First, meaningful labels and their definitions are created and made publicly available (definition step). Second, the labels are attached to each information item. This procedure ensures that communities interacting in the Semantic Web use labels consistently. The Semantic Web defines technologies

that support both steps of semantic tagging. *Ontologies* implement the definition step. The language OWL [90], among others, enables the creation and maintenance of ontologies. The *Resource Description Framework* (RDF) [87] provides a model to express the semantic labeling. It is a graph data model that relates objects to each other by constructing three-tuples (triples) among them. Those triples follow the form (subject, predicate, object). The W3 group coordinates all activities related to the Semantic Web [81]. Figure 2.4 shows the layered architecture of the Semantic Web. We give a short introduction into it.

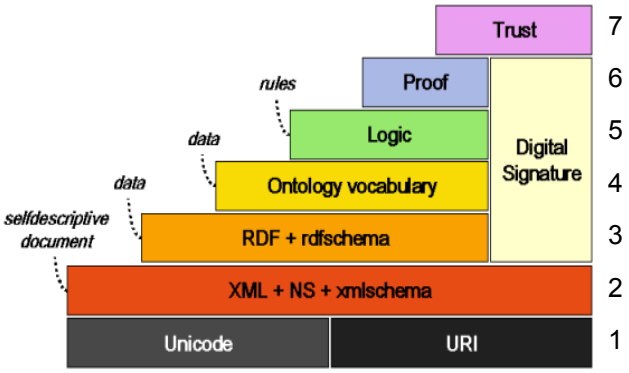


Figure 2.4: The Layers of the Semantic Web [81]

Layer 1: Unicode/URI defines standards for character encoding (Unicode) and resource identification (URI) to allow an access to particular information items.

Layer 2: XML/NS/XML Schema defines standards to exchange structured information (XML, see Section 2.2.2). Also, XML is the exchange format for RDF and OWL documents.

Layer 3: RDF/RDFS defines a graph-based data model to describe resources. It is based on triples (subject, predicate, object). RDFS is the schema language of RDF and allows to restrict RDF documents syntactically.

Layer 4: Ontology vocabulary defines the semantics of tags. It allows the interoperable usage of RDF-documents. Users that share their ontology acknowledge a common understanding of their domain of interest (shared knowledge).

Layer 5+6: Logic/OWL define logic rules using the ontology definition language OWL. This allows to automatically infer new information about the content.

Layer 7: Trust defines methods and technologies to ensure that the content of documents is trustworthy, for example public keys.

Web Services

Although also Web Services provide a framework that eases Web data integration, the focus is different than that of the Semantic Web. We subsume by *Web Services* standards and protocols that allow interoperation of Web applications. Their main focus is the

functional viewpoint of application integration. Web Services help to overcome technical heterogeneities. Main technologies of Web Services include SOAP, WSDL and UDDI.

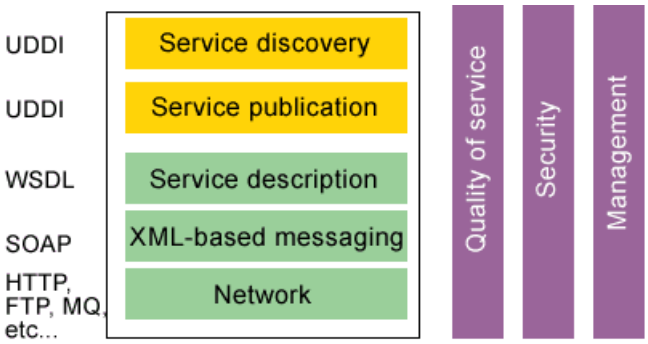


Figure 2.5: Web Service Technology Stack [85]

The **Simple Object Access Protocol (SOAP)** defines a message transfer protocol. SOAP uses XML to describe message properties and content. It substitutes HTML as message format [88].

The **Web Service Description Language (WSDL)** is an XML language that describes interfaces of SOAP-functions [84]. On the one hand, it addresses functional properties (function name, function parameters, data types and result types). On the other hand, WSDL holds attributes about the service itself. Examples are the information about the service location or about the used protocols.

The **Universal Description, Discovery and Integration Service (UDDI)** is a directory service that allows to build up directories where service providers can publish their Web Services. UDDI itself is accessible through SOAP.

2.2.3 Technologies versus Reality

We introduced techniques that have been developed to ease communication on the Web and to overcome problems on syntactical, semantic and technical heterogeneity among systems and data sources. We showed differences to the current Web. Also, we gave an idea about the potential of these technologies when aiming for interoperability and information integration on the Web. A broad-scale application of these sophisticated technologies could solve most of the heterogeneity problems when extracting and integrating Deep Web content. However, we observed that although techniques that enable and ease the programmatic access of the Web are available for several years, the reality shows a sparse utilization of them. Moreover, it is likely that many of the existing Web sources will not adopt the new developments within the next years. Therefore, a large scale integration of Web databases cannot rely on these technologies [4, 19].

We identify two main reasons for this observation:

1. Designers create unstructured content, because they don't consider other than human users.

2. Designers intensionally prevent the programmatic use of the applications, because they don't *want* other than human users.

Unstructured Design The first point addresses "legacy systems" that are common on the Web even though its short history. The main reason that designers of Web sites don't consider a programmatic access is that they are not aware about this opportunity. The publication of content on the Web is a rather creative than a systematic process for many designers. The easy-to-use Web techniques such as HTML enable anybody to provide new contents without any knowledge about structural design. On the one hand, this environment leads to the high popularity of the Web. On the other hand, it prevents a structured, uniform way of accessing data sources and thus it causes problems of heterogeneity that substantially complicate information access and integration.

Conflicts of Interest The second point applies especially in competitive environments, mainly in commercial contexts. There are two reasons for a Web site designer to prevent a programmatic usage: (1) authors intend to secure their applications from illegal usage (for example robots that select data for SPAM distributions) and (2) authors of Web sources want to bind customers to their own service. It is not in the interest of those Web designers to ease comparison and integration of their contents with those of their competitors.

In both cases, designers intensionally construct their Web data sources such that a programmatic exploitation becomes complicated. A popular example for this behavior are embedded images that contain only textual information.

Summarizing, the integration of sources on the current Web demands efforts to ease their programmatic access without relying on designers intentions. Due to many reasons the implementation of an integration system of this kind causes many challenges. We introduce the most important ones.

2.2.4 Challenges of Deep Web Integration

Deep Web sources are *heterogeneous*, *autonomous* and there exists *no common ontology* as a basis for their integration. This results in three main challenges when integrating Deep Web sources:

Content Integration: Deep Web integration systems have to deal with all traditional information integration issues, such as different kinds of schema- and data heterogeneities. The data-driven challenges in a Deep Web context are roughly the same than those when integrating databases [59]. We introduced some ideas of information integration in Section 2.1. For an in-depth insight into this area please refer to [53].

Web Representation Layer: We showed that in many scenarios integration systems can access Deep Web sources only via their representation layer. This presentation

layer is designed for only human interaction. It does not provide meta information about its elements (e.g. attributes). Therefore, additional transformation steps are mandatory prior to the performing of the integration on data level. A Deep Web integration system performs such transformations for example between the text-markup representation of HTML and data-centric representations such as XML. In different contexts this transformation step either is called *extraction* or *wrapping*.

Evolution Awareness: Deep Web sources tend to frequently change their state and appearance. Most of the changes apply only to the presentation layer because designers change the "look&feel" of their application. A robust Deep Web integration approach has be to be aware of changes of both the presentation layer as well as the information structures of the underlying data sources.

2.3 Related Aspects of Deep Web Integration

This work concentrates on query interfaces and their extraction and mapping. It mainly targets virtual integration systems, but parts of the work may be used also in other contexts than the one introduced in Chapter 1. We show interfaces to other approaches to motivate our work in different contexts. Also, our contributions do not cover all steps of Deep Web integration (see Figure 1.3 on Page 3), we therefore show, how different approaches may be used together to provide full-fledged integration solutions. Liu [59] provides a comprehensive description on many Web integration approaches including several algorithm descriptions.

Historically first, the main attention in Web data integration was put on the extraction of structured data from HTML pages. Research concentrated on pages containing data items. In a Deep Web scenario the focus of such approaches is limited to result pages (compare Figure 1.2). A full-fledged integration system may be achieved by combining our contributions with result centric approaches.

Another research direction focused on query processing. On the one hand, systems concentrate on human users and target the construction of a unified query interface for a number of integrated Deep Web sources. On the other hand, the idea of query mediation allows a more flexible integration. A few approaches focus on this issue by reusing of techniques from database integration. These techniques rely on meta information about the used sources. Our contributions may help to provide such meta information.

Another application field are approaches that handle evolution of Web pages, because they rely on extraction and integration techniques. Finally, we shortly introduce crawling-based approaches for the Deep Web. These works collect information stored in Web databases. An efficient crawling through the contents of Web databases demands good meta information about the sources. The first three research directions focus on virtual integration. In contrast crawling-based approaches integrate physically.

2.3.1 Structural Result Wrapping

Result wrapping is the complementary process of query interface extraction. In contrast to the extraction of query interfaces that outputs meta information (e.g. the interface schema), the wrapping of the HTML result pages outputs data entities. Integration systems in the Deep Web context implement wrapper components to transform the unstructured/semistructured HTML results into a set of structured data instances, mainly represented in XML. Please note, that most authors in the Deep Web community use the term *wrapper* only for the result interface processing. Wrappers in this definition do not include parts of query processing as in Wiederhold's definition [83].

The specific approaches used for wrapping of HTML result pages depend on the logical and representation structure of the wrapped pages. We distinguish between *list pages* that contain structured lists of multiple result instances on a single page and *detail pages* that contain only a single data instance. The further is the more common and thus has gained significant research focus. The later has been investigated sparsely.

We briefly describe two techniques for the extraction of list pages [92, 18] and one that is applicable for detail- as well as list pages [21]. For a deeper insight into this topic, please refer to Liu [59].

The main idea of most of the solutions for the result wrapping problem is to find patterns on result pages that help to differentiate content sections of the page from overhead (such as ads, menus, etc.). The idea is driven by the observation that Deep Web result pages are constructed from templates. Thus, the wrapper construction approaches try to re-engineer these hidden page templates.

The IEPAD algorithm [18] and Zhais algorithm [92] are based on alignments of sections of a result page. Whereas IEPAD is string-oriented, Zhais algorithm utilizes the DOM tree. A list page that contains multiple instances should have repetitions in its representing string and its DOM tree. The idea of the two algorithms is to split the HTML-string/DOM-tree into multiple substrings/subtrees, one for each repeated section. The algorithm aligns the substrings/subtrees to each other. Matching regions refer to structural overhead whereas variant sections refer to data instances.

The Roadrunner approach [21] utilizes a related idea, but it is not restricted to list pages. Instead to compare sections on a single page, Roadrunner compares multiple result pages from a single source. It constructs a tree out of each of them and compares the obtained trees. Similarly to the above methods, mismatches refer to data records, matches to HTML structure overhead.

The systems MDR [60] and NET [61] extract data records from flat and nested structured Web pages. Additionally to other approaches they utilize visual information such as the area covered by and the number of data items present in each record.

2.3.2 Building Unified Query Interfaces

A horizontal integration system that focuses on human users demands a *global* query interface that best reflects all the sources integrated. The global query interface provides the single access point to the integrated Deep Web sources and should contain the

most relevant attributes of its source interfaces. The construction of a global query interface corresponds to schema merging in general. It builds up on previously identified mappings.

Dragut et. al. [25, 26] provides an automated solution for the schema merging step of Deep Web integration. Three requirements to an appropriate global query interface are identified: structural appropriateness, lexical appropriateness and instance appropriateness. The structural appropriateness assures that the global query interface reflects the major grouping and ancestor relationships of the integrated query interfaces. The lexical appropriateness assures the right choosing of names such that the hierarchy of attributes is reflected. Finally, the instance appropriateness assures that the domain values in the global interface reflect the domain values of the integrated interfaces. The merging algorithm tries to satisfy all of those requirements. For example, after matching, the elements of the query interfaces to be integrated are clustered semantically. The algorithm assures that each cluster gets an appropriate name.

Recent work [25, 26] has shown that integrated Web query interfaces generated from sources represented hierarchically are qualitatively better than the ones generated from sources having a flat representation [36]. The authors conducted a user survey which showed that interfaces generated from hierarchical representation are easily to understand.

2.3.3 Federated Web Information Systems

A federated Web information system is a virtual integration system that allows complex queries at run-time. It transfers the ideas of a federated information system to a Web context.

We know about two systems that implement a federated information system using Deep Web sources, MetaQuerier [38] and MiWeb [15].

MetaQuerier proposes a system that incorporates all steps toward a federated information system using Deep Web sources. The main distinctive feature to federated information systems is that MetaQuerier performs all integration steps online. During query time it first searches for relevant sources, extracts their schemas, integrates them, distributes the user query to the sources and integrates the results. This system focuses on large scale integration with limited query capabilities. Therefore, it allows no complex query planing like traditional federated systems.

MiWeb [15], in contrast, is a case study that fully implements a federated architecture in a Deep Web context. In its initial configuration MiWeb contains of three sources where one them is a restricted Deep Web source. One issue of special interest investigated in MiWeb is the problem of posing complex queries through a restricted Web query interface (*Query Tunneling*). Query Tunneling [44] simplifies complex queries such that they meet the restrictions of small query interfaces and filters those result entities that do not fulfill the original query after obtaining the results.

2.3.4 Evolution of Web Pages

Evolution awareness is important to cope up with the challenge that Web interfaces change their appearance frequently. There are two different ways to handle the evolution of Deep Web sources. On the one hand, the construction of wrappers and the schema management is separated from the querying process. In this case, a component should monitor interface changes during runtime. On the other hand, the evolution handling problem can be solved by shifting all processes into the online querying step. The further way is represented by classical maintenance systems. They assume that the wrapper has been constructed priorly to the querying process and focus on the result extraction process.

Lerman [51] identified two problems, wrapper verification and wrapper reinduction. The verification system detects when a wrapper is not extracting correct data (caused by a changed Web source). The reinduction system identifies corresponding data on Web pages so that a new wrapper can be generated.

Meng [66] assumes invariant features on a Web page that are preserved while the representation changes. Examples for such features are syntactic patterns, annotations or hyperlinks of the extracted data items. The approach uses these features to identify the locations of the data items in the changed pages and repairs wrappers by inducing semantic blocks from the HTML tree.

Raposo [74, 75] uses labeled example queries for the maintenance of wrappers. The maintenance system collects query results during normal operation. When a source has changed, it uses the stored queries as input to generate a set of labeled examples. These examples help to induce a new wrapper.

An example for the online solution of the evolution problem during query time is the system MetaQuerier [38]. It moves all steps of the integration such as schema management into the online query processing. Therefore, it avoids inconsistencies between its meta information and the current state of the underlying sources. The system does not need a separate maintenance component.

2.3.5 Deep Web Crawlers

Deep Web crawlers try to automatically index Web databases and their contents similarly to classical Web crawlers. They access the Deep Web in a two-step process: (1) Deep Web pages are crawled and stored and (2) user queries are processed against the stored index. The first step is performed offline using a focused crawler [7]. Only the second step is performed online. We distinguish two types of crawling-based approaches: *Surfacing* and *Web Warehouses*. The main difference is the result processing during the crawling step.

Surfacing In this approach, the crawler of a Web search engine tries to pose multiple queries to a Deep Web source through its query interface. The result pages are indexed similarly as pages of the Surface Web and put into a page index of the search engine.

Web Warehouse Whereas the crawling process is similar to surfacing, the result pages are parsed and split into information items. These items build up a data ware-

house. Microsoft implemented a prototype of this approach in the tool Libra [69].

A crawler needs to generate all theoretically possible queries to obtain the complete contents of a Deep Web source when no meta information about the capabilities of its query interface is available. Therefore, this approach is restricted to simple query interfaces. For example, assume a user wants to crawl the content of a used car Web site that contains a query interface with only two fields `Model` and `Zip-Code`. Than, a crawler has to create 32 million distinct queries that result in 32 million form submissions to cover the area and car types of the US. This number is larger than the number of cars for sale in the US [63]. Also, the high network traffic created by this approach disqualifies it from an application in many domains. Therefore, newer Deep Web crawlers [3, 7, 16] put their focus on the understanding of query interfaces. A crawler needs to input meaningful values into the fields of query interfaces to retrieve the data from Web databases [7].

We observed that a preorder traversal of the schema tree of a query interface reflects the way a human being parses the interface in order to understand the meaning of the fields. For instance, in Figure 1.4 before a user reaches the field `From` she first encounters the label `Where Do You Want to Go?`. Thus she has an unambiguous understanding of the meaning of the field `From`. Having a proper annotation of the nodes of the schema tree of a query interface a crawler could follow the same preorder traversal to automatically understand the meaning of the fields within the interface. Another challenge to automatic understanding of query interfaces is the presence of inter-related fields [16]. These fields restrict the kind of queries that can be submitted to a search engine. The problem faced by crawlers is determining those related fields from the Cartesian product of fields. The search space can be significantly reduced as such fields are usually siblings in the hierarchical representation.

Summarizing, brute-force crawling-based approaches are insufficient for Deep Web sources that allow complex queries, that contain huge amount of information, that contain frequently changing information, or that perform complex calculations to get results to a particular query. Instead, a crawling of these sources demands meta information about the capabilities of their query interfaces to reduce the number of queries.

2.3.6 Mashups and Entity Search

We understand by *mashup* a Web application that combines data or functionality from multiple sources in order to enable value added services. Mashups follow ideas of the Service Oriented Architecture (SOA) which understands the Web as large repository of services that may be accessed and combined using standardized techniques [29]. An example implementation of a mashup tool is Yahoo Pipes[3].

Most mashups concentrate on simple keyword queries but there are recent developments toward more complex application scenarios. Hornung et. al. [41] describes general challenges of mashups using Deep Web sources and focuses on the problem of *chaining* queries of multiple sources by executing them consecutively using the results of one data source as input parameters of a second data source. The authors identify six challenges:

[3] see www.yahoo-pipes.com

form interaction, *data record extraction and labeling*, *fuzzy result lists*, *data cleaning*, *assisted mashup generation* and *combinatorial explosion*. Summarizing, in a Deep Web context mashup-tools analyze query interfaces as well as result interfaces of the used data sources and combine their contents meaningfully.

Thor et. al. [80] concentrates on a general architecture to mash up arbitrary heterogeneous sources while Endrullis et. al. [28] investigates search strategies in complex querying scenarios by using *query generators*. The presented approach is based on the idea of entity search engines. Entity search engines focus on the retrieval of particular business entities (such as Flights in the Airline domain) utilizing predefined entities. Therein, query generators are able to automatically determine a set of search queries to retrieve entities from an entity search engine by using the given set of entities. Therefore, a query generator for an entity search engine has to cope up with similar problems than Deep Web crawlers.

Given the above classification of challenges, our contributes could improve mashup tools mainly during query interface analysis. For example, the identified mappings among the query interface elements may be used to reduce the combinatorial explosion when chaining queries.

3 Query Interface Extraction

The success of many applications in the Deep Web hinges upon a good *understanding* of Web query interfaces, because a query interface provides a glimpse into the schema of the underlying database and is the main means to retrieve data from the database.

We emphasized in Section 2.3 that applications such as Deep Web crawling and Web database integration require a programmatic access to Web query interfaces. Therefore, an important problem to be addressed is the automatic extraction of query interfaces into an appropriate model.

Besides identifying all its fields, the understanding of a query interface includes: (1) grouping the fields into semantically connected sets, (2) tagging fields and groups with their semantic roles and (3) annotating fields and groups with additional meta-information (e.g. data type). `Departure Date` is an example of such a group in Figure 3.1. Groups can form bigger building blocks on the interface, e.g., the `Departure Date` and `Return Date` groups form the block `When Do You Want to Go?`. Such grouping naturally leads to a hierarchical representation of query interfaces. Second, tagging assigns *labels* to fields and groups. For instance, in our running example the text `Adults` is assigned as a label to the field `Adults` and the text `Number of Passengers` is assigned to the group of fields `Adults` and `Children`. Third, information such as data type and unit of measurement must be determined.

Such an extraction is challenging because query interfaces are represented in HTML, lack a formal specification and are developed independently (compare Section 2.2.4). HTML is a markup language, designed for formatting documents and not to express data structures and semantics. HTML has a rather loose grammar and browsers do not enforce the grammar when displaying HTML pages. As a result, ill-written HTML pages can often be displayed by browsers and used by people. Query interface design seems rather heuristic in nature—there is no clear guidance of how to create an interface. Interfaces follow different design patterns (e.g., the orientation of labels can vary—the fields in Figure 3.1 have the labels above while the fields in Figure 3.2 have the labels to the left). Furthermore, similarly looking interfaces can be developed with different HTML constructs.

The goal of this chapter is the development of an algorithm that extracts and maps query interfaces into a hierarchical representation. We hypothesize the existence of a set of domain-independent "commonsense design rules" that guides the creation of Web query interfaces. This hypothesis along with visual cues transform query interfaces into schema trees. Consequently, we describe a Web query interface extraction algorithm, which combines HTML tokens and their layout positions within a Web page. Tokens are classified into several classes out of which the most significant ones are text tokens and field tokens. A tree structure for text tokens is derived using their geometric layout.

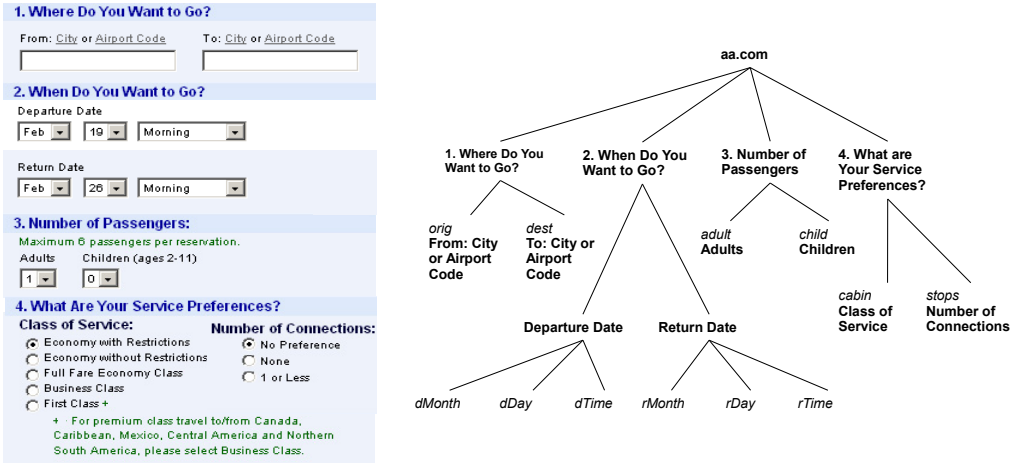


Figure 3.1: An example of an ordinary query interface in the airline domain along with its schema tree.

Another tree structure is derived for the field tokens. The hierarchical representation of a query interface is obtained by iteratively merging these two trees. Thus, we convert the extraction problem into an integration problem.

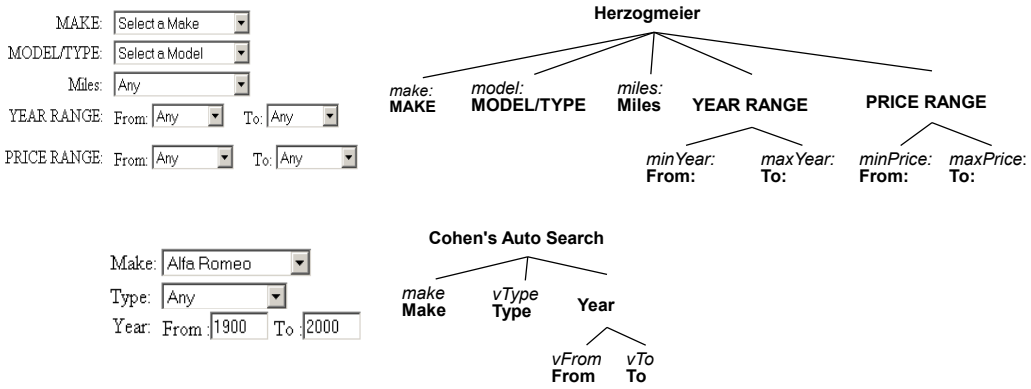


Figure 3.2: Two query interfaces from the auto domain.

In the remaining sections of this Chapter, we first define our understanding of query interfaces and introduce fundamental commonsense rules for their design in Section 3.1. The main extraction algorithm is defined in Section 3.2. We describe the evaluation of the algorithm in Section 3.3 and discuss related work in Section 3.4.

The results of the query interface extraction process are schema trees (compare Definition 2.5 on Page 22). We will utilize them to identify semantic correspondences between query interfaces in Chapter 4. These correspondences may later be used for multiple purposes during consecutive integration steps. We exemplify one use case in Chapter 5 when

describing an approach for the classification of interfaces by their application domain.

3.1 Fundamentals

We now outline our understanding of Web query interfaces. We describe elements and structural properties of Web query interfaces and define commonsense design rules for Web query interfaces. The extraction algorithm later exploits these design rules.

3.1.1 Representation of Query Interfaces

In this section, we present our way of modeling a query interface (see Definition 2.2). Methods for mapping a given Web query interface into such a model are described in the rest of the Chapter.

Query Interface Elements

Data in searchable databases are accessible through form-based search interfaces (mostly HTML forms). The basic building blocks of these forms are: *text input boxes, selection lists, radio buttons* and *check boxes*. We will generically call them fields. A *text input box* is rendered as an empty box with or without a default value. The Field `From` on the interface in Figure 3.1 is such a field. A *selection list* presents the user with a set of choices to select from. There are two types of selection lists: single selection list (e.g. combo box) and multiple selection list (e.g. listbox). *Radio buttons* and *check boxes* are employed by designers to explicitly display the choices to the user. For example, in Figure 3.1 the `Class of Service` is shown as a group of radio buttons. The difference between radio buttons and check boxes is that choices are exclusive in a radio button group, whereas multiple check boxes may be selected at the same time. A radio button group can be regarded as a single selection list and a group of check boxes as a multiple selection list. For example, the set of radio buttons denoting the class of service is treated as a field (single selection list) whose label is `Class of Service`. The labels attached to each individual radio button (e.g. `Economy with Restrictions`) become the values of the selection list field. To summarize, we have two types of fields: fields with predefined sets of values and fields without predefined values.

A field has a *name*, which identifies the field in the HTML script (for programming purposes). Fields may also have *labels* that describe to users the meaning of the field. Fields may not have their own labels, rather they share a group label with other fields. For instance, the three fields denoting the departure date in Figure 3.1 do not have their own labels but they share a group label. In some cases the label may be entirely left out as the designer relies on the set of values of the field to convey the semantics of the field.

While names are readily available from HTML, the assignment of labels requires substantial work, but is necessary for the correct understanding of the semantics of a field or group of fields. Beside of fields, we identify also complex elements. Complex elements emerge from a group of fields that forms a semantical unit.

Structural Properties

Since query interfaces provide access to Web databases they have to visually convey information about the data in the databases so that the user can easily infer the necessary data he/she has to provide in order to obtain the desired information. An important aspect of user interfaces is a sort of *spatial locality property* among the fields. That is, semantically related fields are usually *grouped* together in an interface. For example, in the interface in Figure 3.1 the fields denoting the types of passengers traveling are next to each other. Moreover, several related groups can further be grouped together. In the interface in Figure 3.1 the two groups of fields denoting departure date and return date, respectively, are put together (`When Do You Want to Go?`). Thus, this bottom-up characterization gives rise to a hierarchical structure for interfaces. In addition, each group of fields may have labels that describe to the user what the group is about.

The hierarchical structure of query interfaces was first observed by [91]. More in detail, a query interface is an ordered tree of elements so that leaves correspond to the fields in the interface, internal nodes correspond to groups of fields in the interface, and the order among the sibling nodes within the tree resembles the order of fields in the interface (this is from left-to-right for documents in the western world). This schema tree (see Definition 2.5 on Page 22) captures both the order semantics and the nested grouping of the fields in a query interface. Figure 3.1 shows a typical example of a query interface in the airline domain and its corresponding schema tree. Observe that the schema tree has four levels and that each level except for the root refines the information in the level above. The first level is a generic root node (usually the root has the name of the Web site).

Three more types of meta-information are attached to the leaves (fields) and internal nodes (groups of fields): *domain type*, *default value* and *unit of measurement*. Domain types are of two kinds: simple and complex. Examples of the former type are integer or string. Examples of the latter type are date, time, datetime and currency. Complex domain types can be associated with a group of fields (e.g. `Departure Date` in Figure 3.1 has the datetime domain type) Many fields have default values, e.g., the fields denoting month and day of the departure date in Figure 3.1. Frequently a default value may appear in a text box. As shown in [16] the default value of a text box may be a valuable indicator of the kind of input the field expects, since the domain type of text boxes is difficult to determine in general. Units of measurements such as `miles` and `square feet` are important pieces of information that need to be properly extracted and attached to the right field. They frequently appear abbreviated in query interfaces (e.g. `mi` stands for `miles`). The abbreviations are recognized by consulting certain Web sites, e.g. `www.abbreviations.com`.

3.1.2 Commonsense Design Rules

In this section, we describe our observation that almost all real-life Web interfaces obey to a small set of rules that partly determine their appearances. Although they appear trivial in first place, we shall show in Section 3.2 that exploiting these rules enormously

helps in re-engineering Web interfaces in a formal model.

Automatic extraction of query interfaces is challenging because interfaces are created autonomously and with languages (e.g., HTML) obeying a loose grammar. The question arises whether there is an inherent set of rules that designers of query interfaces intuitively follow. Our investigation of a reasonable large number of query interfaces in various domains showed that a *small* set of *commonsense design rules* emerges from heterogeneous query interfaces. We first enumerate the rules and then motivate them by drawing a parallel between documents and query interfaces.

Definition

Except for Rule 0 and 6, all the other are new, and not encountered in any of the previous extraction techniques. This is of no surprise since none of them have the concepts of groups and subgroups.

- **Rule 0:** Query interfaces are organized top-down and left-to-right.
- **Rule 1:** Fields within an interface are organized in semantic units of information, i.e. groups.
- **Rule 2:** A label is used to denote either the semantics of a field or of a group of fields, but not both.
- **Rule 3:** If a field f with a label l_f belongs to a group g with label l_g then the *text-style* of the label l_f is different from the text-style of label l_g.
- **Rule 4:** If a group g with a label l_g is a subgroup of a group G with label l_G then the text-style of the label l_g is different from the text-style of label l_G.
- **Rule 5:** The labels of all the members of a group have the same text-style.
- **Rule 6:** The orientation of a label of a field is either to the left, above, right or below of the field. The label of a group is either above or to the left of the group.
- **Rule 7:** The labels of all the members of a group have the same orientation.
- **Rule 8:** Let G be a group and g be one of its subgroups. Suppose a label with text-style FS_1 is assigned to G and a label with a different text-style FS_2 is assigned to g, then for any group H and its subgroup h the label assigned to H cannot have the text-style FS_2 when the label assigned to h has the text-style FS_1.

The first two rules phrase rather obvious observations. First, ordinary people expect the content of documents (Web pages) to be laid out in a predicted pattern—i.e. top-down and left-to-right. Second, the content of such a document must be structured in some organic units so that it is easy to understand—e.g., it would be rather peculiar to have the fields denoting the departure date separated by some other fields, such as the passenger fields. Rule 2 says that a label cannot play multiple roles as this would confuse an ordinary user.

In a text document, such as this document, headings are employed to organize the content of the document. Headings are located at the top of sections and subsections which they delimit. Headings serve several important roles in documents: they preview and succinctly summarize upcoming content and they show subordination. They naturally lead to a hierarchical structure for a document.

In many ways a Web query interface can be regarded as a document: its fields along with their labels are the content and the labels of the groups are the headings. A label of a group of fields, similarly to a heading, should succinctly summarize the upcoming set of fields. For example, the label `Where Do You Want to Go?`, in Figure 3.1, describes the purpose of the fields in the section it introduces. A user, thus, learns that the fields `From` and `To` represent the departure and arrival information, respectively.

A set of heuristics emerges from the parallel between query interfaces and documents. The text-style of a heading is different from the text-style of the content. Likewise, the label of a field has a distinct text-style than that of a label of a group of fields (Rule 3). The text-style of a heading is different than that of its subheadings; similarly, the text-style of the label of a group is distinct from that of the label of its subgroup (Rule 4). Headings at the same depth in the document hierarchy have the same text-style and labels denoting sections at the same depth of a query interface have the same text-style (Rule 5). The subheadings of a heading have all the same alignment (e.g., left, center). Likewise, the labels of the members of a group have the same orientation (Rule 7). If a text-style is chosen for a heading H and a different text-style is chosen for its subheading h, there must be no other subheading having the text-style of H and moreover there must not be a sibling heading of H having the same text-style as h. If this rule is translated to the labels in a query interface, Rule 8 is reached.

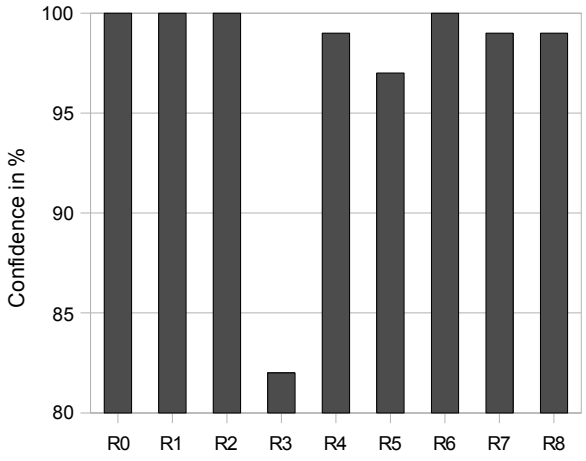


Figure 3.3: Histogram of rule confidence.

Evaluation

We conducted an informal survey of these rules. The ICQ dataset was used[1]. The histogram in Figure 3.3 depicts the outcome of our study. The survey reveals that there are many similarities between the organization of a document and that of a query interface. All the rules, except for Rule 3, are satisfied by almost all query interfaces in the dataset. Rule 3 is violated in 18% of all the interfaces. The reason is that these interfaces use the same text-style for the labels of fields as well as for the labels of groups. An example is shown in Figure 3.2, the field-label `Make` utilizes the same text-style as that of `Year Range`. We introduce a heuristic to cope with such cases in Section 3.2.3.

The survey also shows that this set of rules holds across diverse domains (ICQ has interfaces from five domains). It implies that there are implicit conventions that influence the design of Web query interfaces. Although there is not a common wisdom how to build an ideal query interface, the creative process is guided by the way most humans expect documents to be laid out. People in the western world read from left to right and top to bottom. Furthermore, objects referred throughout a document are a priori defined. A group of fields (section) on a query interface can be regarded as such an object and its label as its definition. Consequently, when humans visually parse query interfaces they expect to encounter the label before the group (section). These rules act as axioms to build the data structures employed in the extraction algorithm.

The set of rules is by no means universal. The rules can be easily adapted to accommodate query interfaces developed for people speaking languages following other orientation patterns. We manually inspected query interfaces intended for Arabic languages. We observed that these interfaces are organized from right to left and top-down. For these interfaces, one only needs to swap "left" with "right" in the commonsense rules.

3.2 The Extraction Algorithm

This section describes the algorithm for extracting query interfaces. First, we give a high level description of the steps of the algorithm. Each is explained in detail in the following sections. These are the steps of the algorithm:

Token Extraction: An HTML Page is input into a rendering engine of a browser (e.g. IE). A list of *tokens* is extracted from it. A token is an atomic visible element on the page. The token list is cleaned and filtered. There are three types of tokens considered: *text tokens*, *field tokens* and *image tokens*. Each token is enclosed in a rectangular area that describes the layout coordinates of the token in the actual window frame. This area is called *bounding box* (see Definition 3.3 on Page 44).

Tree of Fields: An initial tree of fields, called FT, will be generated based on the order and alignment of the fields in the rendered version of the interface. Fields and groups correspond to leaves and to internal nodes, respectively, in the tree. Additionally, a set of candidate labels is determined for each field. The tree in Figure 3.5 represents FT derived from the interface in Figure 3.1.

[1]The datasets used in this work are described in Section 3.3

Figure 3.4: Fields and texts with bounding boxes.

Definition 3.1 (Tree of Fields) *Given a set of fields F, we define the ordered Tree of Fields FT such that (1) each leaf of FT is a field and (2) consecutive fields that are visually aligned on the Web page will be grouped together. Each leaf in FT is represented by the field name, it additionally may have a label and value sets.*

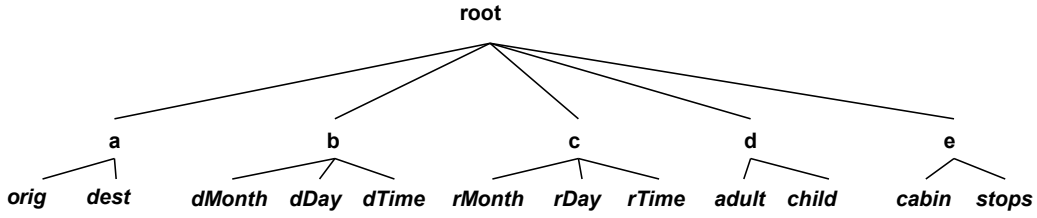


Figure 3.5: Tree of fields.

Tree of Text Tokens: We hypothesize that a text token in an interface has a *semantic scope*. Intuitively, this is the area of the interface which is characterized by the semantic meaning of the text token. We later define the semantic scope more precisely (Definition 3.6 on Page 50). As an example, the semantic scope of `When Do You Want to Go?` is the rectangular area that includes every text token and fields that are between the text token itself and the text token `Number of Passengers` (Figure 3.6). The semantic scope of `Departure Date` includes the text token and the three fields denoting month, day and time of departure. The tree of text tokens, called TT, is inferred from the inclusion relationship between the rectangular areas defining the semantic scopes of text tokens (e.g., the semantic scope of `When Do You Want to Go?` includes that of `Departure Date`, thus the latter text token is a child of the former text token). The tree in Figure 3.7 represents the tree of text tokens derived from the interface in Figure 3.1.

Definition 3.2 (Tree of Text Tokens) *Given a set of text tokens T and their attached semantic scopes we define the ordered Tree of Text Tokens TT such that (1) each node of TT corresponds to a text token in T and (2) the tree structure is derived from the inclusion relationship of the text tokens semantic scope.*

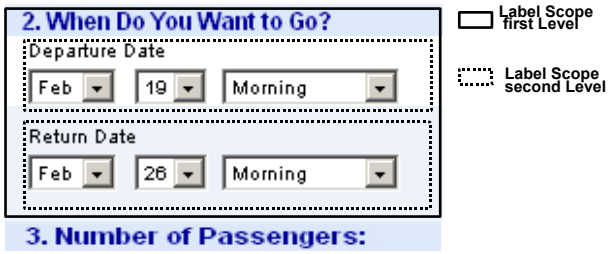


Figure 3.6: Semantic scopes of text tokens.

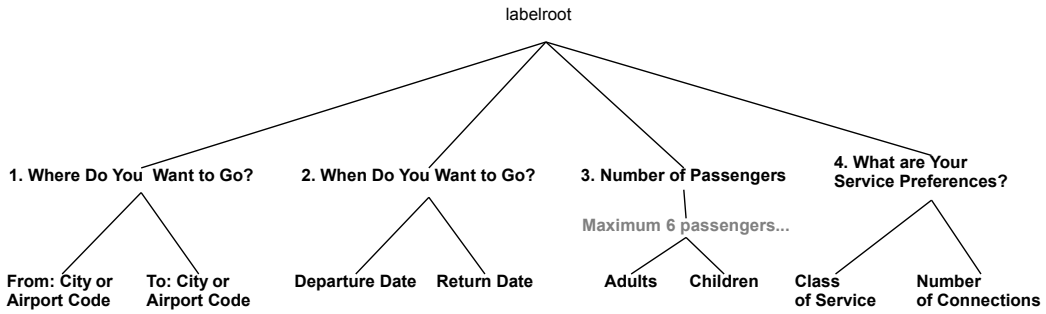


Figure 3.7: Tree of text tokens.

Integration: The final hierarchical representation ST (schema tree) of the interface is obtained by merging the two trees FT and TT. FT is the target tree and TT is the source tree. A directional "mapping" from TT to FT is defined. A label is mapped into a leaf (field) if it was determined to be a candidate label. The semantic scope of a label l contains a set of fields. A label is mapped into an internal node if its semantic scope contains all the fields of the internal node. Multiple labels may be mapped into each node of the tree. New internal nodes may be added to the tree of fields FT. The goal of this step is to find the final schema tree and the assignment of labels to its nodes. For our running example, the final schema tree ST is depicted in Figure 3.1, on the right.

We are now going to describe all introduced steps of the algorithm in detail.

3.2.1 Token Extraction

We subsume by token extraction all preprocessing steps of the algorithm. The result is a list of tokens. The token extraction process consists of three main tasks: preparation, transformation and cleaning of an HTML page that contains a query interface.

Preparation

Before performing any transformation on the raw data we prepare the HTML source such that it follows certain rules. Beside of the correction of syntax-errors we separate texts from tags such that the later transformation steps easily can construct a tree from it.

HTML allows multiple ways to insert texts among tags that lead to different DOM trees. In general, fields and adjacent texts may not belong together. Therefore, we first treat texts and fields independently. We normalize the HTML such that each piece of text is enclosed into surrounding `<SPAN>`-tags and separated from adjacent tags. For example, the HTML sequence `<td>From:<input name=orig>To:<input name=dest></td>` is transformed to: `<td> <span>From:</span><input name=orig> <span>To:</span> <input name=dest></td>` The example underlines the intention of this preprocessing step: In the originating HTML snippet both texts are assigned to a single DOM node, in the transformed snippet each of them is aligned to a unique node.

Transformation

In this step, we transform the HTML source into a set of *tokens*. Tokens are built up from *bounding boxes* that enclose visible elements on a Web page:

Definition 3.3 (Bounding Box) *A bounding box* $bb = [(x_1, y_1), (x_2, y_2)]$ *is a rectangular area described by two absolute pixel coordinates of its upper left corner* (x_1, y_1) *and lower right corner* (x_2, y_2) *in the browser window where* $x_1 \leq x_2 \wedge y_1 \leq y_2$.

For example, a text of the width of 100pt and height of 20pt that is aligned in the upper left corner of the browser window has the corresponding bounding box $[(0, 0), (100, 20)]$.

We now define a token as follows:

Definition 3.4 (Token) *A token t is a relation of an atomic visible element on a HTML page (either a field, text or image) and its bounding box. Depending on the type of the element, the token may carry particular metainformation about the element (e.g. style-instructions). The style information may contain the attributes font-style, font-size, font-color, background-color, font-family and font-weight.*

Corresponding to their content tokens are classified into three types: *text tokens*, *field tokens* and *image tokens*. Text tokens represent pieces of texts inside the interface. In the running example, e.g. all labels such as `Adults`, or `When Do You Want to Go?` as well as comments, e.g. `Maximum 6 passengers per reservation`. Field tokens are extracted from each field of the interface, such as text boxes or radio buttons. In Figure 3.1 the text box `Adults` as well as each radio button (e.g. the radio button in front of the text `Business Class`) is a field token. Note that at this step of the extraction only the fields itself are considered as field tokens, whereas their labels are separate text tokens.

In contrast to most other approaches (e.g. [37]) the algorithm does not rely on a HTML parsing but uses the output of a rendering engine in order to obtain a list of HTML tokens. It does not exploit the DOM tree to achieve the semantic relations of the elements but calculates them using visual clues such as bounding box inclusions.

Cleaning

We call the cleaning step token processing, because this step prunes the number of tokens by employing multiple heuristics. The raw token list may contain noisy data mainly for two reasons: (1) structural noise and (2) semantical noise. Structural noise may occur because tokens may be duplicated or contained into each other in the underlying DOM structure. Moreover the list of raw tokens contains also tokens which are not in the area of the `<form>`. These tokens need to be pruned. Semantical noise appears mainly to text tokens. Ideally all final text tokens are candidates for labels or values. But in reality there are multiple texts on a Web interface which do not belong to these class (e.g., comments, descriptions, ads). These will be pruned in the preprocessing step. Finally all remaining text tokens are normalized in order to make them comparable. More in detail the token processing contains of the following tasks:

Removal of out-of-form Tokens Tokens which are not inside the actual form are removed by checking the inclusion relationship of the bounding box of the form token and the bounding boxes of each other particular token.

Removal of Buttons Tokens inside the actual form which are not necessary for the extraction process (buttons, hidden fields, etc.) are removed.

Removal of non atomic Text-Tokens Non atomic tokens contain other tokens. Only tokens which do not contain other tokens remain. For instance the HTML-snippet `<span><span>Text1</span><span>Text2></span></span>` is transformed into three tokens (one for the outer `<span>` and one for each of the two inner `<span>` tags. The outer token is a non atomic token and is removed.

Removal of Comments We subsume by comments all texts which do not refer to a label or value. We use several heuristics to identify comments. The most important are that only comments may contain negations, may start with parentheses, may contain examples or they may include the terms `and/or`.

Resolution of Name Conflicts Names identify fields but some real interfaces contain distinct fields sharing the same name. These fields are renamed.

Splitting of Text Tokens containing more than one Label In some interfaces multiple labels are combined into one text token (e.g., the labels are divided by spaces). These tokens are split into multiple tokens such that each token describes one logical label.

3.2.2 The Tree of Fields

This section describes the methodology for the construction of the tree of fields of a query interface. Recall that the main goal is to infer the "hidden" schema tree structure of a Web query interface. Two issues need to be addressed. First, given that the schema tree of a query interface is an ordered tree, the problem is finding the *semantic order* of the fields on a query interface. The semantic order of the fields on a query

interface is the order in which a user reads and fills in the fields. On our running example (Figure 3.1), the semantic order of the fields is `From, To, Departure Month,..., Number of Connections`. The second problem is the grouping of related fields. The third problem is the assignment of candidate labels to the leaves of the schema tree.

Semantic Order of Fields

For each field a `tabindex` attribute can be set in HTML. The tabbing order defines the order in which elements receive focus when navigated by the user via the keyboard. Designers *may* employ this attribute to specify the fill in order of fields in a query interface. This attribute is scarcely used. Only 10 out of 100 interfaces in the ICQ dataset utilize it. The question remains, whether there is any other way to infer the fill in order of the fields when the `tabindex` is not specified for the fields. Our empirical study shows that the order in which the fields are encountered in the source code usually coincides with the fill in order. This is also the strategy employed by the rendering engines of browsers. The explanation is that, while developing Web query interfaces, designers place fields in the HTML source code in the order the user parses them. To conclude, whenever `tabindex` is encountered in a Web query interface we use it to determine the semantic order of fields and when it is not present we utilize the order the rendering engine provides.

Grouping of Related Fields

Although the HTML language and its add-ons have some constructs (e.g. `<fieldset>`) that could be used to emphasize certain grouping of fields in a HTML page, which in turn could be employed to infer the hierarchical structure, our experience with real world Web query interfaces shows that these constructs are sparsely found within the source code of Web pages. But, are there any other layout hints that could be employed to infer groupings of fields?

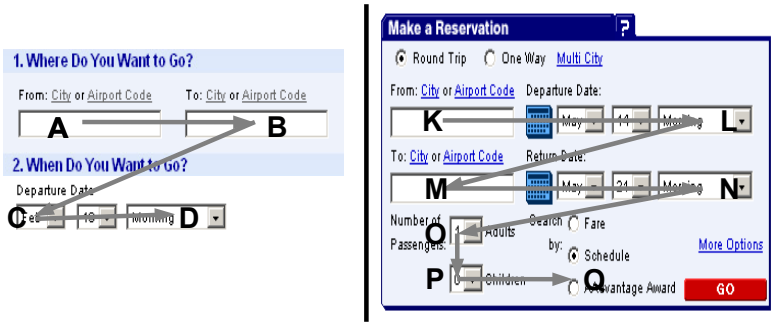


Figure 3.8: Semantic order and inflection points.

We noticed the following geometric pattern for Web query interfaces. If straight line segments are drawn between any two consecutive fields in the semantic order, then all

these line segments form one connected curve. The curve consists of horizontal, vertical and diagonal line segments. A horizontal (vertical) line segment corresponds to a set of fields laid out row-wise (column-wise) on the visual rendering of the query interface. Figure 3.8 (left half) shows the curve for a fragment of the interface in Figure 3.1. The curve may also have a number of *inflection points*.

Definition 3.5 (Inflection Point) *An inflection point is a point on a curve where the curve changes from being concave upwards (positive curvature) to concave downwards (negative curvature), or vice versa. An inflection point marks either the end or the beginning of a semantic group of fields in the interface.*

Thus, in our case, an inflection point is a point where two non-parallel line segments meet. For instance, in Figure 3.8 (left half) inflection point B marks the end of the group of fields `From` and `To` and point C marks the beginning of the semantic group `Departure Date`. An inflection point is the geometric "evidence" that the designer finishes/begins the organization of a subset of fields into a group of semantically related fields. Hence, a horizontal/vertical line segment emphasizes the presence of a group of fields. For example, the line segment $[C, D]$ denotes the group `Departure Date` while the vertical line segment $[O, P]$ in the right half of Figure 3.8 represents the group `Number of Passengers`.

Once the groups of fields have been determined, the tree of fields is constructed bottom up as follows. We start with a flat tree—all fields are children of the root and ordered from left to right according to their semantic order. Then, an internal node is added to the tree for each determined group of fields. Figure 3.5 shows the derived tree of fields for the running example (e.g., an internal node was added for the groups of fields `From` and `To`).

There is one more issue to be addressed: Can a field belong to multiple groups of fields? In our geometric interpretation this corresponds to an inflection point that joins a horizontal with a vertical line segment. On such an occurrence the field is assigned to the group corresponding to the horizontal line segment since the fields on query interfaces are mostly row-wise organized.

Candidate Labels for Fields

Another important problem in the construction of the schema tree is the semantic tagging of its leaves (fields) and internal nodes (group of fields). There is no consistent pattern across query interfaces as to where a label is positioned with respect to the field it defines. A label may be to the left, to the right, above or below of the field. This enumeration is the order of places a label is most likely to appear for a field. One may choose to assign labels according to this order. This is the strategy employed by [93, 37], but it is not generic and leads to errors. Our strategy instead is to collect first all possible candidate labels for each field and then, in a later step (integration, see Section 3.2.4),

to decide which are the appropriate ones. More in detail, first, for each field, a set of candidate labels is found from all text tokens, using Rule 6. This reduces the set of text tokens which are possible labels for the field. Then, the groups of fields are computed. Afterward, a text token is chosen from the set of candidate labels as the final label, if it satisfies Rules 5 and 7. These steps are applied iteratively, as whenever a final label is determined for a field, the set of candidate labels for every other field or group of fields is updated (Rule 2).

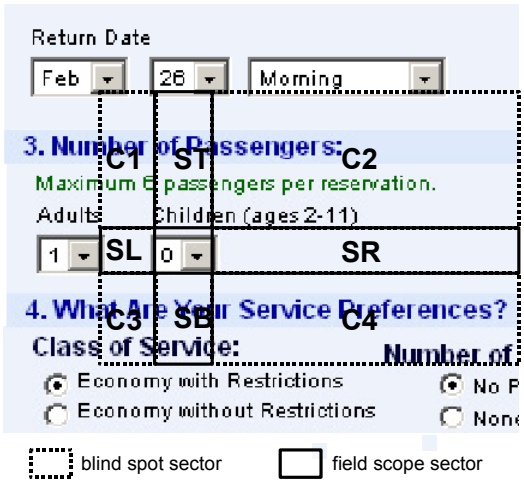


Figure 3.9: Exemplified field scopes.

We now consider the possible locations where the label of a field should be placed with respect to the bounding box of the field. This is called the *scope of the field*. The space around a field is partitioned into 8 sectors by the vertical and horizontal lines going through the top-left and bottom-right corner points of the bounding box of the field. Figure 3.9 shows the sectors around the field `Children` (the combo box having the value 0 selected). The corner sectors, marked with C_1 to C_4 in the figure, are called *blind spots*, because the label of a field cannot reside *entirely* in those areas. If the label were entirely in one of these sectors, then between the label and the field would be a diagonal relationship. This would be a violation of the common sense Rule 0, that information is organized top-down and left-to-right. The four sectors, denoted ST (top sector), SL (left sector), SR (right sector) and SB (bottom sector) constitute the scope of the field. A candidate label must lie in one of these four areas, although it might extend into a blind spot. For example, the scope of the field `Children` is surrounded by solid lines in Figure 3.9. Each sector is a bounded rectangular area. Each sector starts at the bounding box of the field and stretches in one of the four directions (e.g., top sector stretches upward) until the bounding box of another field or the boundary of the interface is met. A candidate label is a text token whose bounding box intersects the scope of the field. For each sector an ordered list of candidate labels is computed. The order is given by the distance between the field bounding box and the candidate label

bounding box. In our example, the left and right lists of candidate labels of the field `Children` are empty. The top list consists of `Children, Maximum 6 passengers per reservation` and `Number of Passengers`. The bottom list consist of `What are Your Service Preferences?` and `Class of Service`.

3.2.3 The Tree of Text Tokens

The second piece of information employed for determining the hierarchical structure of a query interface is the layout relationship between text tokens, which is covered in this section. We distinguish between *text tokens* and *labels*. The former refers to any text appearing in a query interface, e.g., comments. The latter is a text token that was identified to be the semantic tagger of a field or a group of fields.

The hierarchical structure of headings is easy to extract for documents because headings are explicitly tagged. The labels in Web query interfaces lack such tagging, thus the problem of their hierarchical subordination is harder. Nevertheless, using the analogy between headings and the labels assigned to a group of fields, a technique can be designed. There are two main observations. First, headings at the same depth in the heading hierarchy of a document have the same text-style. Consequently, we expect that the labels assigned to the groups at the same depth in the schema tree of a query interface to have the same text-style (Rule 5). Thus, the first task is to cluster/classify the text tokens appearing on a query interface based on their text-style properties. Second, for any two headings H and h, with h subheading of H, the content area covered by h is a subset of that of H. So, if we knew the (semantic) area each label covered on the interface then the hierarchical subordination relationship between the labels in a query interface could be determined by using the inclusion relationship. We show how the semantic areas of text tokens are estimated and used to infer the hierarchical relationship between them in this section.

Clustering of Text Tokens

Each text token has a complex style attribute, which describes its layout properties: font-color, background-color, font-size, font-style, font-weight and font-family. The properties are retrieved from the rendering engine of a browser. The text tokens with the same values for their style properties are clustered together. For our running example (Figure 3.1), there are three clusters of text tokens:
$C_1 =$ { `Where do you want to go?`, `When...`, `...`};
$C_2 =$ { `From, To, Departure Date, ...`};
$C_3 =$ { `Maximum 6 passengers per reservation, ...`}.

A preprocessing step is required before the clustering. There are instances when the label we seek to find is split into multiple text tokens and each token may have a distinct text-style, for example, the four text tokens `From, City, or` and `Airport Code`. A designer may choose to construct labels in this manner to emphasize certain keywords in the label. For a proper clustering we need to concatenate these *consecutive* tokens and to assign them a unique text-style. Two text tokens t_1 and t_2 are consecutive if their

bounding boxes are aligned horizontally and the right coordinate of the bounding box of t_1 equals the left coordinate of t_2. A new token t is created such that the bounding box of t is the union of the bounding boxes of t_1 and t_2. The text-style of t is obtained from the text-style of t_1. The process is applied until all consecutive text tokens are concatenated. In our example, the final text token is: `From City or Airport Code`.

Semantic Scopes of Text Tokens

Since the relationships between a label l of a group of fields and the objects it characterizes are not explicitly given in the HTML source code, we hypothesize that the semantics of l summarizes a (rectangular) area on the visual rendering of the interface. Thus, any object in this area is semantically described by the label. Next, we describe our method for estimating the areas characterized by labels in a given query interface.

We define formally:

Definition 3.6 (Semantic Scope) *The* semantic scope *of a text token t, denoted scope(t), is the maximal rectangle with the following properties:*

1. *Its left-upper corner coordinates are the coordinates of the left-upper corner of the the bounding box of text token t.*
2. *It extends downward and to the right until the semantic scope of another text token p in the same style cluster or the boundary of the interface is met.*
3. *Let q be a text token from a different style cluster than that of t. If the bounding box of t is included in the semantic scope of q then the semantic scope of t is included in the semantic scope of q.*

The semantic scope of the text token `Departure Date` (Figure 3.6) starts at the left-upper corner of the bounding box of the text token and continues downwards until text token `Return Date` is met and to the right until the boundary of the interface. Its semantic scope is included by the semantic scope of the text token `When Do You Want to Go?`, because its bounding box is inside the semantic scope of `When Do You Want to Go?`.

In order to estimate the semantic scope of a text token t on the visual rendering of a query interface we need to know the text tokens of the same text-style "closest" to it in either rightward or downward directions. We call them *rightward neighbor*, t_r, and *downward neighbor*, t_d, respectively. The boundary of the semantic scope of t is computed with respect to its neighbor text tokens or the boundary of the interface.

Definition 3.7 (Rightward Neighbor) *Let $\mathcal{T}$ be the set of all text tokens of a specific style in a given interface and $t, p \in \mathcal{T}$. Further let $(p.X_1, p.Y_1)$ and $(p.X_2, p.Y_2)$ be the*

left-upper and right-bottom coordinates of the bounding box of p. The rightward neighbor t_r of the token t is defined:

$$t_r = \min_{p.X_1}\{p|p \in \mathcal{T} \wedge p.X_1 > t.X_2 \ \wedge \ p.Y_1 > t.Y_1\}$$

Definition 3.8 (Downward Neighbor) *Let $\mathcal{T}$ be the set of all text tokens of a specific style in a given interface and $t, p \in \mathcal{T}$. Further let $(p.X_1, p.Y_1)$ and $(p.X_2, p.Y_2)$ be the left-upper and right-bottom coordinates of the bounding box of p. The downward neighbor t_d of the token t is defined:*

$$t_d = \min_{p.Y_1}\{p|p \in \mathcal{T} \wedge p.Y_1 > t.Y_2 \ \wedge \ p.X_2 > t.X_1\}$$

Definition 3.7 says that from the set of all text tokens in the same style cluster as t that reside to the right of t and are not above of it, the text token with the smallest X_1-coordinate is the rightward neighbor. In a similar way, Definition 3.8 defines the downward neighbor. It is possible that the boundary of the interface is met, a small adjustment is made to each definition. In our running example, the downward neighbor of the text token `When Do You Want to Go?` is the text token `Number of Passengers` (Figure 3.1) and the rightward neighbor is the right boundary of the interface.

Having these concepts defined we can provide the algorithm for the computation of semantic scopes and the tree of text tokens. The function for computing the hierarchical relationship between text tokens is depicted in Algorithm 3.1. The algorithm is recursive. The input consists of the current rectangular window frame WF on the visual rendering of a query interface. The output is the root (artificially created) of the tree of text tokens. The algorithm is initially called with the window frame of the entire query interface. In the current window frame, the algorithm first retrieves the set of text tokens *topT* in the same style cluster satisfying the properties: (1) the union of their semantic scopes covers the entire set of fields in the current window frame and (2) it is the smallest set out of all such sets of text tokens. This set of tokens is called *top level tokens*. The procedure `ComputeTopLevelTokenSet` finds this kind of tokens and is shown in Algorithm 3.2.

Then, for each token t in the set *topT* the following steps are performed. First, token t is appended as a child to the current root. Second, the semantic scope of token t is computed using the neighbor text tokens of t, Definitions 3.7 and 3.8 (procedure `getSemanticScope`). Third, the set of text tokens that are inside of the window frame defined by the semantic scope of t is retrieved. Finally, the algorithm is recursively called with the window frame defined by the semantic scope of t. The recursive call terminates when there are no text tokens *topT* in the current window frame.

Algorithm 3.1 ComputeTokenTree($WF, root$)

Input: current window frame WF
Output: the root of the tree of tokens

```
topT = ComputeTopLevelTokenSet(WF, root);
for all t ∈ topT do
  create node node_t for token t;
  root.addChild(node_t);
  scope_t = getSemanticScope(t, WF);
  ComputeTokenTree(scope_t, node_t);
end for
```

Now we describe the steps of the procedure `ComputeTopLevelTokenSet`. The algorithm commences with the retrieval of the set of fields F and the set of text tokens T which are in the current window frame WF. Then, it classifies the text tokens T according to their text-styles. For each cluster of text tokens c, the set of fields that is included in the union of the semantic scopes of the tokens in c is computed (procedure `getCoveredFieldSet`). If there are clusters of text tokens whose semantic scopes cover the entire set of fields F (in the current window frame WF) then the cluster with the smallest number of text tokens with this property is returned. Going back to our analogy with headings in a document, at a higher level in the headings hierarchy fewer headings are needed to cover a portion of the document than at a lower level of the hierarchy. In our example (Figure 3.1), the fields in the second section are contained in both the union of the semantic scopes of the text tokens `Departure Date` and `Return Date`, and the semantic scope of the text token `When Do You Want to Go?`. Hence, the latter is picked because it uses fewer tokens to cover the fields in the section. (Figure 3.6).

The ELSE branch handles those instances of query interfaces that do not obey the analogy with documents. The first exception is when fields are not in the semantic scope of any text token within the current window frame. This may happen when fields or groups of fields have no label. This case is illustrated by the set of radio buttons `Round Trip, One Way` in the query interface in Figure 1.1 (A) on Page 2. This field has no label. (Recall that a group of radio buttons is regarded as a single selection list field whose values are the labels attached to the radio buttons.) Thus, it is not in the semantic scope of any text token. Fields that are not in the semantic scope of any text token are removed from the set of all fields. The algorithm starts a new iteration with the remaining set of fields (outer WHILE-Loop). If no single cluster covers the set F of fields in the current window, but the set is covered by text tokens of different text styles, the ideal solution would be to find the minimum set of tokens whose union of semantic scopes contain all the fields in F. Since each semantic scope of text token has a subset of fields from F a semantic scope can be regarded as a subset of the powerset $\mathcal{P}(F)$. The set of all semantic scopes becomes a family $\mathcal{S}$ of $\mathcal{P}(F)$. Therefore, our problem becomes finding the subfamily $\mathcal{C} \subseteq \mathcal{S}$ of sets whose union is the set of fields F and is the minimal subfamily with this property. It can be shown that this is the same as the well-known NP-hard set cover problem[20]. So, we are left with devising an approximation

solution. We implemented the greedy algorithm for the set cover problem, embedded in the procedure `applySetCoverHeuristic` which is not shown in detail.

We show how the tree of text tokens (Figure 3.7) for the running example is obtained by the algorithm `ComputeTokenTree`. The initial window frame is the entire query interface. The text tokens of each of the first two clusters C_1, C_2 cover all the fields of the interface. The text tokens of cluster C_3 do not cover fields such as the one with label `From` and are therefore discarded. Since the cluster C_1 has a smaller number of text tokens, its tokens become the first level of the tree. Then, the semantic scope of each token in C_1 is computed. For example, the semantic scope of `When Do You Want to Go?`, called SC, stretches all the way to the right boundary of the interface and down to the token `Number of Passengers`. SC includes the text tokens `Departure Date, Return Date`. The algorithm recursively computes the semantic scope of each of these two text tokens. The union of the semantic scopes of `Departure Date` and `Return Date` includes all the fields within SC. Therefore, these text tokens become the children of the text token `When Do You Want to Go?`. Furthermore, the algorithm recursively goes into the semantic scopes of these two text tokens. Since there are no other text tokens in their scopes, this branch of the recursive calls terminates. All other subtrees are obtained similarly. The resulting tree is shown in Figure 3.7.

Adjustments

As shown in the histogram in Figure 3.3, Rule 3 is not always satisfied. The main issue is that designers choose to assign labels with the same text-style to both fields and groups of fields. For instance, the labels for fields `From` and `To` have the same style as the label `Year` in Figure 3.2 on the right. The semantic scope of `Year` is "empty", because its rightward neighbor is `From` and its downward neighbor is the boundary of the interface. Therefore, its scope contains no field. To overcome this issue, we expand the semantic scope of such a text token to the right and downward until one of the following conditions applies: (1) the next token with an empty semantic scope, (2) a text token with at least two fields in its scope or (3) the interface boundaries is met. The expanded semantic scope of `Year` contains the fields with labels `From` and `To` due to the satisfaction of Rule 3.

3.2.4 Integration

Given the tree of fields FT (e.g., that in Figure 3.5), the tree of text tokens TT (e.g. that in Figure 3.7) and the lists of candidate labels for the fields of a Web query interface we develop an algorithm to derive an integrated tree structure that represents the schema tree ST of the interface (e.g., that in Figure 3.1). The body of the algorithm is given in Algorithm 3.3 on Page 55. The tree of text tokens is used to determine labels for the internal nodes (group of fields) of the schema tree and to prune the initial candidate labels of fields as determined in Section 3.2.2.

Using the semantic scopes of text tokens, the set of candidate labels of a field can be further pruned. If a field f is within the semantic scope of some text token t, then any

Algorithm 3.2 ComputeTopLevelTokenSet $(WF, root)$

Input: the window frame WF, the root of subtree
Output: set of tokens in the same style cluster
whose semantic scopes cover WF

```
F = getInsideFields(WF);
T = getInsideTokens(WF);
while T ≠ ∅ do
  C = getStyleClusters(T);
  for all c ∈ C do
    compute the set of fields covered by the tokens in c;
    F_c = getCoveredFieldSet(c);
  end for
  if there is a cluster c ∈ C : F_c = F then
    Let C' be the set of these clusters;
    return min_|c|{c ∈ C'}
  else
    F' = getNonCoveredFields(F);
    if F' ≠ ∅ then
      F = F − F';
    else
      F'' = getSingleCoveredFields(F);
      if F'' ≠ ∅ then
        T'' = getCoverTokens(F'');
        root.addChildren(T');
        F = F − F'';
        T = T − T'';
      else
        T = applySetCoverHeuristic(F, T);
        return T;
      end if
    end if
  end if
end while
return ∅
```

Algorithm 3.3 TreeIntegrator(FT, TT)

Input: Tree of fields FT, tree of labels TT
Output: Schema tree ST

```
ST = FT
prune candidate label sets of leaves in FT using TT;
discard text tokens from TT whose semantic scopes contain no fields;
assign text tokens to fields explicitly specified in HTML;
while no more changes to ST do
  for all sets of sibling nodes in ST do
    assign labels to leaves according to Rules 5 and 7;
  end for
  ST = doMerge(TT.root,ST);
end while
ST = postProcess(ST, TT);
```

candidate label for f is t, one of its descendant text tokens or null. For example, in Figure 3.6 the field `dTime` (the first field having default value `Morning`) cannot have the text token `When Do You Want to Go?` as a candidate label from its upward direction, because the field is within the semantic scope of `Departure Date` and `When Do You Want to Go?` is not a descendant of `Departure Date`. Similarly, candidate labels of a field from the other directions can be pruned.

The labels for the fields are determined from the remaining candidate labels. The algorithm first assigns those text tokens as labels to fields that are explicitly specified by designers in the HTML code (tag `<label>`), if such specifications exist. Although, HTML provides this method to explicitly associate fields with their labels, we noticed that it is rarely used (in less than 10% of the interfaces in the dataset). The algorithm also discards the text tokens from the tree of tokens, whose semantic scope does not contain any field. Then, the algorithm iteratively performs two tasks. It assigns labels to leaves according to Rules 5 and 7. That is, the labels of the fields in a group have to be on the same side of the fields and they have to have the same text-style. The algorithm may discover new groups of fields in addition to those already found in the tree of fields and assigns, whenever possible, labels to internal nodes. This step is accomplished by procedure `doMerge` (Algorithm 3.4 on Page 57). These steps are performed as long as there are *changes* to the schema tree. There are two types of changes: new internal nodes are added to the schema tree ST (i.e., new grouping of fields) or labels are assigned to the nodes in ST.

The schema tree may require additional adjustments. In a final step, `postProcess` (procedure not shown here), the algorithm may further adjust the structure of the tree and may assign additional text tokens to nodes. This step handles those fields which should not form a group but are incorrectly placed within a group in the tree of fields (described in Section 3.2.2). On some interfaces, this is caused by fields which are grouped horizontally by its designer for the sole purpose of optimizing the space occupied

by the interface on the Web page. We observe that such a group on this kind of interfaces has the following characteristics: it contains only two fields, does not have labels and the interface is entirely constructed out of such groups. The parent of these fields is removed from the schema tree ST and the grandparent becomes their parent.

Second, the body of the algorithm assigns a set of candidate labels to each internal node. For a set of sibling internal nodes the labels are chosen according to Rule 5, i.e., the labels have the same text-style.

Procedure `doMerge` (Algorithm 3.4) recursively integrates the tree of text tokens into the tree of fields. The target tree is the tree of fields and the source tree is the tree of labels. The initial schema tree is the tree of fields. The output is the tree resulted from the integration of the two trees. In a preorder traversal of the tree of text tokens TT the algorithm performs the following operations. A node v with text token T in the tree of text tokens is mapped into a node lca of the schema tree. The node lca represents the lowest common ancestor of the set of fields contained in the semantic scope of the text token T (procedures `mapFieldList` and `getLCA`). Three cases are handled based on the relationship between the set of descendant leaves D_{lca} of the node lca and the list of fields, S_F in the semantic scope of the token T represented by node v. (i) If $D_{lca} = S_F$, the text token T is assigned as a candidate label to lca. (ii) If $S_F \subset D_{lca}$ and the fields in S_F are children of lca, then a new node is inserted in the schema tree as a child of lca. The children of lca in S_F become children of the new node. (iii) If $S_F \subset D_{lca}$ and and the fields in S_F are *not* all children of lca, then descendant leaves of lca in S_F are reorganized. A new internal node is created and the fields in S_F become its children. The new node is inserted among the children of lca such that the semantic order of the first field in S_F is preserved. Note that this reorganization of the leaves may be inconsistent with the initial semantic order of the fields. This occurs when the groups constructed from the semantic order of fields are in contradiction with the groups suggested by the semantic scopes of text tokens. As a rule of thumb, whenever such a contradiction occurs, the semantic scope of the label provides the intended way the fields should be grouped.

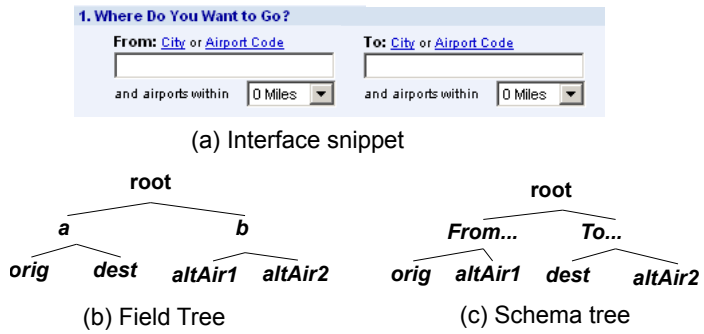


Figure 3.10: Example interface containing a group contraction.

This is illustrated on the fragment query interface in Figure 3.10 (a). The initial

Algorithm 3.4 doMerge(node,FT)

Input: the root of the tree of text tokens, schema tree ST
Output: integrated schema tree ST

```
V = node.getChildren;
for all v ∈ V do
  T = mapFieldList(v);
  lca = getLCA(T, ST);
  D_lca = getDescendentLeafSet(lca);
  if D_lca = T then
    v candidate label for lca;
  else
    if all t ∈ T children of lca then
      lca.appendChild(T);
    else
      w = lca.createChildInOrder(T);
      w.addChildren(T);
    end if
  end if
  doMerge(v,FT);
end for
```

tree of fields suggests the groups `orig` and `dest`, and `altAir1` and `altAir2` (See tree (b)). The semantic scope of `From` indicates that the field `orig` should be grouped with `altAir1`. Following this instead of the groups inferred from the semantic order leads to a better organization of fields as shown in tree (c).

The result of integrating the two trees in our running example is shown in Figure 3.1, on the right.

Note that the schema tree produced by the algorithm `TreeIntegrator` has the following property. If label l_v is assigned to node v and label l_w is assigned to node w with v an ancestor of w then label l_v is an ancestor of l_w in the tree of text tokens TT. This result guarantees that Rules 4 and 8 are satisfied by the labels assigned to the groups and subgroups of fields in the final schema tree of a query interface.

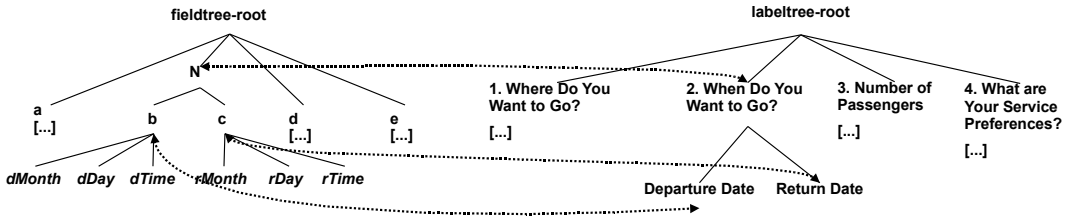


Figure 3.11: Integrating the tree of fields (left) and the tree of labels (right)

In our running example, the tree of fields of the interface in Figure 3.11 has an internal

node, denoted by `b`, with children `dMonth`, `dDay` and `dTime` (these fields do not have labels, their names in HTML source code are used instead and they denote departure month, day and time, respectively) and another internal node `c`, with children `rMonth`, `rDay` and `rTime` (representing return month, day and time, respectively). In the tree of labels, `Departure Date` and `Return Date` are children of `When Do you Want To Go?`. The semantic scope of the label `Departure Date` contains the fields `dMonth`, `dDay` and `dTime`. Hence, the label `Departure Date` is mapped to the node `b`. Similarly, the label `Return Date` is mapped to the node `c`. Moreover a new node `N` is inserted into the tree of fields. This node is mapped to the label `When do you want to go?`

3.3 Experimental Evaluation

We have implemented an operational prototype extraction system. We have conducted extensive experiments over several domains of Web sources to evaluate our approach. Our study intends to evaluate whether our solution can be used for arbitrary application domains, ranging from simple query interfaces to nested, multi-field forms, and whether it substantially improves on previous work.

3.3.1 Datasets

Three datasets were considered:

The **ICQ** dataset consists of query interfaces in five domains: airline, automobiles, books, jobs, real estate. Each domain has 20 query interfaces, so in total there are 100 interfaces.

Tel8 is the dataset employed in [93]. It consists of 487 query interfaces, of which we could use only 50% because the others refer to no longer existing Web servers. For these interfaces the browser either crashes or displays no page. Interfaces are from eight domains: airlines, auto, books, car rentals, hotels, jobs, movies and music records.

The ICQ and Tel8 datasets are publicly available[2].

WISE is the dataset used in [37]. It consists of 147 interfaces, out of these we were able to work with 134 due to the same reason as mentioned for the Tel8 dataset. The interfaces of this dataset come from seven domains: books, electronics, games, movies, music, toys and watches. Overall, we evaluated our approach on more than 500 web interfaces from 15 distinct domains. The used interfaces differ largely in terms of number of fields, depth of nesting, layout, etc. Table 3.1 and Table 3.2 give some figures on the ICQ and WISE datasets.

3.3.2 Performance Metrics

We evaluated our algorithm according to different metrics, concentrating on the difficult tasks in interface extraction. In each metric, we compared a particular type of information obtained automatically from our method with the "true" information as defined

[2] see the University of Illinois at Urbana Champaign Web Repository `http://metaquerier.cs.uiuc.edu/repository/`

Domain	Leaves			Internal Nodes			Depth		
	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
Airline	10.95	1	18	5.3	1	7	2.5	1	4
Automobiles	4.95	2	10	1.85	1	4	1.4	1	2
Books	5.45	1	12	1.85	1	5	1.45	1	2
Jobs	4.55	1	7	1.45	1	5	1.25	1	2
Real Estate	6.95	1	18	3.05	1	10	1.8	1	4
Overall	6.57	1	18	3.75	1	10	1.68	1	4

Table 3.1: Characteristics of the ICQ dataset.

Domain	Leaves			Internal Nodes			Depth		
	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
Books	4.73	1	22	1.73	1	8	1.41	1	3
Electronics	5.71	1	14	1.79	1	4	1.36	1	2
Games	5.56	2	18	1.78	1	7	1.22	1	2
Movies	5.08	1	24	1.4	1	7	1.16	1	3
Music	4.12	1	11	1.2	1	3	1.16	1	2
Toys	5.13	3	6	1.25	1	3	1.13	1	2
Watches	6.85	1	25	1.77	1	5	1.38	1	2
Overall	5.08	1	11	1.57	1	8	1.29	1	3

Table 3.2: Characteristics of the WISE dataset.

in a gold standard. How these gold standards were obtained is described below. The metrics are:

Leaf Labeling: The problem of finding the right label for leaves (fields) is difficult because labels are not explicitly assigned to fields (as highlighted previously). We compute the ratio of the number of correctly labeled fields to the total number of fields (accuracy).

Schema Tree Structure: This measure aims to quantify the distance between the structure of the extracted schema tree and that of the ideal schema tree while ignoring the labels assigned to the nodes. It shows how well the groups of fields are identified and how accurate the semantic order of fields is obtained. We use as measure the *structural tree edit distance* which is the minimum number of operations (insert, delete) to convert one tree into another. The precision per interface is $P_s = (N_e - D_s)/N_e$, where N_e is the number of nodes in the extracted tree and D_s is the structural tree edit distance. The recall per interface is $R_s = (N_e - D_s)/N_i$, where N_i refers to the number of nodes in the gold standard tree. Finally, we compute the F-score $F_s = 2P_sR_s/(P_s + R_s)$.

Overall Metric: This measure aims to quantify how well the trees along with their labels are extracted. Therefore, we use the *tree edit distance*, i.e., the minimum number of operations (insert, delete and relabeling) to convert one tree into another. Precision, recall and F-measure are defined as above.

For the latter two metrics, we shall report the overall precision, recall and F-score obtained by averaging over all interfaces in a dataset. Note that neither metric counts the retrieval of domains, data types and default values, because these are easily retrieved from the HTML code.

Gold Standard: Our extracted results are compared against gold standard interface representations. The ICQ dataset already provides the gold standard. For the WISE dataset, we manually constructed the gold standard. For the Tel8 dataset, we only compare the extracted labels for fields and groups (called *conditions* in Tel8) as specified in its gold standard. Note that our algorithm actually identifies many more important groupings, but these cannot be evaluated directly on the Tel8 gold standard.

3.3.3 Evaluation of the Algorithm

Figure 3.12 summarizes the results of our experimental study on the ICQ and WISE dataset. From left to right, the bars represent the accuracy of retrieving the right labels for fields, the F-score for the evaluation of the structure of the schema trees and the overall F-score. The identification of labels for leaves reaches 92% accuracy in both datasets. We also achieve very high accuracy for the structural extraction (>94%). Results for the overall measure are slightly worse (87.5% for ICQ and 91% for WISE), which indicates that sometimes labels for internal nodes are missed. The frequency of this problem correlates with the accuracy of the structural extraction. For instance, if the descendant nodes of an internal node in the schema tree are not properly identified, the label for that internal node may be missed. Thus, the differences in the performance between WISE and ICQ can be boiled down to the differences in the structural accuracy which again depends on the complexity of the schemas (see Tables 3.1 and 3.2).

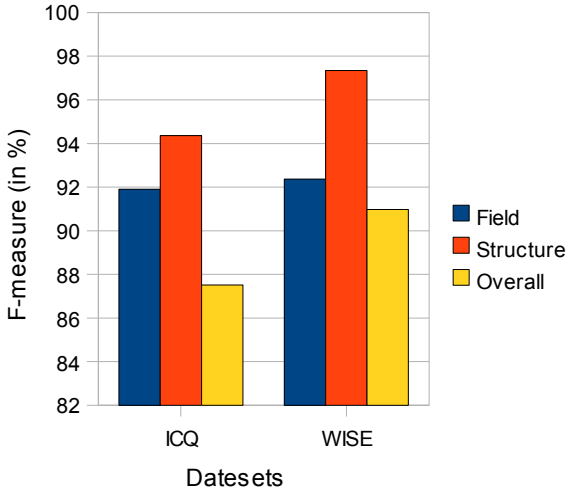


Figure 3.12: Experimental results.

We also ran the experiment for the Tel8 dataset. The overall measure is 87.5%. Note however, that this number gives only a partial impression of the performance of our algorithm on this dataset because the gold standard offers less information than extracted by our algorithm.

Evaluation of the Commonsense Rules. The pie chart in Figure 3.13 shows the influence of the different rules when being run on the ICQ set. Each slice represents the ratio of the total number of times the rule was used to produce labels over the total number of usages of any rule. Obviously, all rules were necessary to achieve a proper result. Except for Rules 1 and 6, the influence of the rules is roughly the same, which suggests that this dataset contains a balanced number of flat and hierarchical interfaces.

Efficiency of the Algorithm. We also evaluated the efficiency of the system. The entire system has two parts, rendering/loading the Web page and the extraction of the query interface. While the rendering takes on average 4 seconds per Web page, the extraction itself is efficient; it takes on average 1 second per user interface.

3.3.4 Discussion

In this subsection we describe the experimental comparison of our system with others. Also, we characterize limits of our solution.

Comparison with other Systems

We compared the performance of our method with that of the WISE Extractor[37] and the system from [93] (abb. as BEP: best effort parser). Since the structures extracted by the three algorithms can not be compared directly, we resorted to a simpler evaluation

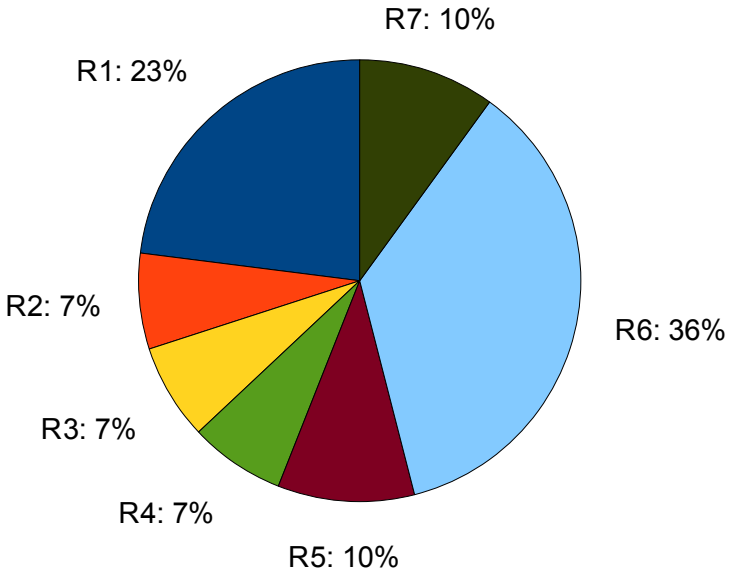


Figure 3.13: Relevance of commonsense rules.

by computing only the accuracy for field and internal node extractions. This measure was also used in [37, 93]. Note that it disregards the ability of our tool VisQI to extract deeply nested structures.

WISE extractor. We thank the authors of [37] for sharing code and dataset. We run WISE Extractor and our tool VisQI on all three datasets. Figure 3.14 shows average results for all datasets. VisQI on average is slightly better on the WISE dataset, but much better on the other two sets: 90% vs. 80% for ICQ, 88% vs. 82% for Tel8. Over all three datasets, the difference in accuracy between VisQI and WISE extractor is 6.5% on average. Our explanation for the poorer performance of WISE on Tel8 and ICQ is that these datasets have more complicated interfaces (see Tables 3.1 and 3.2) than those in WISE dataset. For instance, the airline domain has on average 5.3 internal nodes whereas none of the domains in the WISE dataset exceeds on average 2 internal nodes. Furthermore, ICQ and Tel8 were collected over several years. During this period, the HTML language itself has evolved, which poses problems to parsing-based techniques as the WISE Extractor is a parsing based technique. The WISE-Extractor is also unable to distinguish between visible and invisible fields on a query interface. In comparison, our visual technique employed in VisQI is more resilient to language evolution and does properly handle invisible elements.

BEP. We were not able to obtain the system described in [93]. Consequently, we can only compare to the results published in this paper, and only on the Tel8 dataset. We obtain an overall accuracy of 88% whereas [93] reports an accuracy of 85% [93].

Thus, although BEP uses a set of rules that were directly derived from the dataset it was evaluated on, our system - using a small and generic set of extraction rules - outperforms this method on its own dataset.

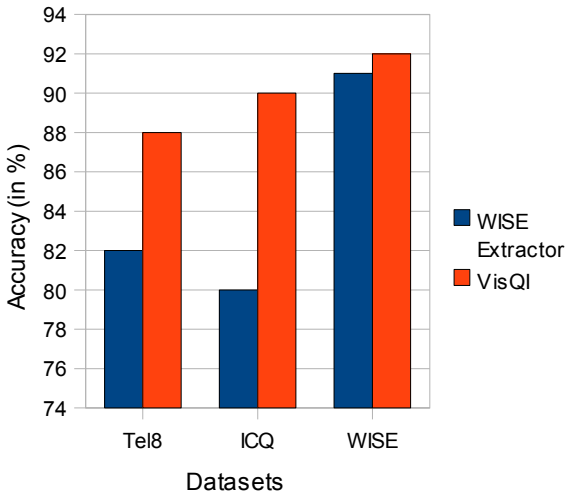


Figure 3.14: Experimental comparison.

Limits of the Approach

Overall, we reach very high accuracy values over a wide range of interfaces and domains. However, one can still strive for improvements. We manually investigated those interfaces where we performed poorly and found a number of problematic situations. First, there are interfaces whose labels are encoded in images. These interfaces account for about 2% of the investigated query interfaces. Although there are tools to extract text from an image (e.g. OCR) our current implementation does not include them. About 1% of the interfaces have labels that are aligned to the center and not to the left. The semantic scopes of this kind of labels are wrongly computed. A number of query interfaces (less than 1%) have a different semantic order than the fill in order which our algorithm expects. Many of the remaining errors in the extraction process root in the preprocessing step of the algorithm. For instance, during tokenization a single text token obtained from the browser may contain the labels for multiple fields.

3.4 Related Work

Hierarchical Web interface models are an important step toward a better integration of Web sources. We believe our approach is unique in that it produces structured schema trees which offer a rich set of information for integration systems.

There have been a number of recent suggestions to improve the extraction of Deep Web interfaces. The approaches presented in [45, 72, 48] regard query interfaces as a flat collection of fields. Consequently, the main problem addressed is the prediction of the right label for a field. One frequently used heuristic employed for prediction is the textual similarity between the name of a field and a label. In the example of Figure 3.1 the name of the field `Adults` is `numAdults` and its label is `Adults`. Since `numAdults` and `Adults` share a significant text portion their similarity score is high enough to suggest that the latter is the label of the field. In our initial effort, we implemented this heuristic as well, but we noticed that its prediction accuracy is rather minute. Moreover, it is not able to discover labels for groups of fields (internal nodes) since these do not have names/ids within HTML pages. Subsequently this heuristic was dropped from our implementation. Furthermore, a flat representation of interface fields fails to properly represent the semantic relationships between them.

The approaches most related to ours, in that they also view query interfaces as structured objects, are [93, 37]. [37] uses *attributes* to group sets of related fields. For example, the three fields denoting the departure date in the interface in Figure 3.1 would be captured as one attribute, and the field `From` on the same interface would be an example of an attribute containing a single field. However, attributes cannot be nested, and therefore this approach would fail to see that the fields `From` and `To` should be grouped together and that they can be semantically characterized by the label `Where Do You Want to Go?`. Similarly, [93] proposes to use a form of grouping fields, but is not capable of representing arbitrarily nested structures. In contrast, our fully hierarchical approach describes Web interfaces as schema trees.

Another feature of our approach is the exploiting of visual layout information for recovering the structure of an interface and for assigning labels to fields. Most existing tools for Web wrapping are based on HTML parsing instead. However, HTML is a rather fuzzy language with many ways to express the same appearance. Especially with newer HTML technologies (e.g., style sheets), the neighborhood of tags and text in HTML is blurred completely and thus does not provide robust clues for interface extraction. In contrast, our method uses visual techniques to analyze Web pages and thus is more robust against HTML particularities and code evolution. The work most similar to our approach is [93] which introduces the notion of viewing query interfaces as visual languages. However, their extraction algorithm uses a grammar with more than 80 productions that was manually derived from a single corpus of 150 interfaces. When using such a high number of very specific rules there is a probability that the resulting system may overfit to the corpus the rules were derived from [67]. Moreover the rules derived in [93] reflect concrete patterns found on Web interfaces at the time the system was developed. But the appearance of Web interfaces evolves with new opportunities of the underlying technology. Therefore, our algorithm is based on only 9 general rules which are completely domain-independent and do not rely on specific specific implementation issues.

4 Query Interface Matching

Most Web database integration scenarios require the knowledge about the semantic overlappings among the schemas, because this information enables to compare the elements of the underlying databases. It is fundamental for many sophisticated techniques such as distributed query processing or the construction of integrated views (or query interfaces) over multiple databases.

The problem is, that most Web databases do not provide their schema descriptions or other meta information that easily enable the identification of semantic overlappings. Therefore, techniques that infer this semantic overlapping information from the publicly available interfaces of Web databases are of particular importance when focusing on information integration in the Deep Web.

A survey [19] estimated interfaces of different domains and found that, for instance, the 50 investigated airline interfaces used altogether less than 100 distinct elements. Most of the interfaces contained around 10 elements. Among the 100 elements only a very small number appeared on almost all investigated interfaces, but most were shared by at least two interfaces. In general, the total number of elements in query interfaces in a particular application domain may be large, but only a few of these elements occur frequently. Additionally, investigations such as [19] have shown, that the elements on Web query interfaces reflect many relevant attributes of the underlying databases. These observations imply that an integration of Web databases based on semantic overlappings of query interface elements is promising. We call such semantic overlappings on element level *mappings*. In contrast, we use the term *matching* for the set of all mappings between elements of two query interfaces and as a name for the process of its computation.

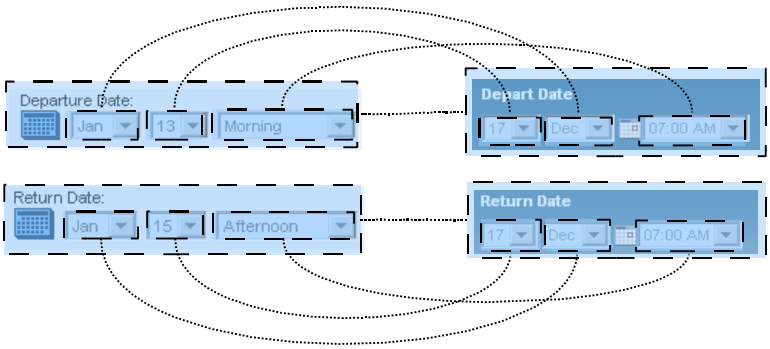


Figure 4.1: Examples of query interface element mappings.

Figure 4.1 exemplifies snippets from the two interfaces of our motivating example

(compare Figure 1.4 on Page 7 and Figure 1.5 on Page 7). It emphasizes mappings between the elements of both query interfaces by dotted lines. Altogether, these mappings form a matching between the two interfaces.

In this chapter, we focus on the identification of binary element correspondences and the derivation of those valid interface matchings that best reflect the semantic overlappings of the interfaces. Hereby, we use ideas of natural language processing [64], information retrieval [2] and graph theory [12].

The chapter is structured as follows. First, we give a broad overview about our matching framework in Section 4.1. We describe the preprocessing step of constructing a normalized dictionary of interface terms in Section 4.2. A set of measures that allows to compute a similarity score between interface elements is described in Section 4.3. We identify candidate mappings by using the measures and compute interface matchings from them. We describe the matching algorithm that takes as input a set of interfaces in their schema tree representation and outputs a set of mappings in Section 4.4. Section 4.5 evaluates our matching algorithm using a gold standard of 140 interfaces from seven domains. We discuss specifica and limits of our algorithm in Section 4.6. An overview about the related work in the area of query interface matching finally is given in Section 4.7.

4.1 The Matching Framework

We propose a method that semantically matches query interfaces based on semantic one-to-one correspondences of their elements. In order to establish interface matchings, we first compute similarity scores between their elements. We obtain the similarity score for each particular pair of elements from multiple properties that characterize query interfaces. We distinguish between *local similarities* and *structural similarities*. Local similarities reflect properties of a single query interface element whereas structural similarities reflect the tree structure and the order of its nodes. Each element pair is a mapping candidate.

Next, we transform the problem of identifying a matching between two query interfaces to a bipartite graph matching problem. Bipartite graphs have two disjoint node partitions. All edges connect elements of the two partitions. The nodes inside each partition are not connected. We model the problem of matching two query interfaces in a bipartite graph as follows. Each partition of a bigraph represents a single query interface. The elements of the two query interfaces correspond to the nodes of the graph. Mappings between elements of two query interfaces are edges between a particular node in the first partition and a particular node in the second partition. The precomputed element similarities correspond to weights on these edges. A matching in a bipartite graph is a configuration of edges that do not share nodes. This property of graph matchings corresponds to the desired one-to-one relationship of element mappings between query interfaces.

We therefore define an interface matching as follows:

Definition 4.1 (Interface Matching) *Let $S = s_1, ..., s_m$ and $T = t_1, ..., t_m$ be the sets of elements of two query interfaces and G be a bipartite graph constructed from these element sets such that each partition corresponds to the elements of a single query interface. Let further E be the set of edges of G. Then, a valid matching M of S and T is defined as graph matching of both partitions S and T:*

$$M \subseteq E \wedge \forall (s_i, t_i), (s_j, t_j) \in M : s_i = s_j \Leftrightarrow t_i = t_j$$

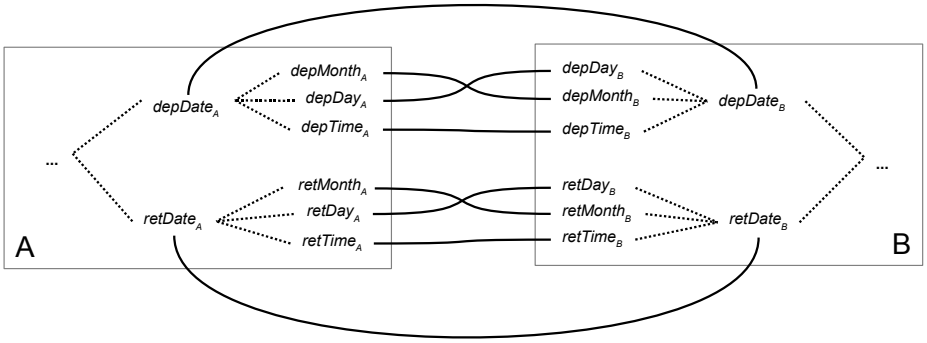


Figure 4.2: Example of an interface matching.

Figure 4.2 shows a valid matching for the portion of the interfaces in our motivating example as introduced before in Chapter 1. For example, the field $depDay_A$ becomes a node in the graph, the mapping $(depDayA, depDayB)$ an edge. There are eight mappings (solid lines). Please note, that edges in the figure showing the tree structure (dotted lines, for example, $(depDate_A, depDay_A)$) are not part of the matching.

In the given example, all edges together form a valid matching. In our understanding, a valid interface matching may contain only disjoint edges (compare Definition 4.1). The correspondence of query interface matchings and graph matchings ensures the strict one-to-one property for all element mappings.

We may derive multiple matchings between two query interfaces that follow Definition 4.1, but most of them are not optimal for our purpose. As introduced before, the main aim of query interface matching is to identify most of the semantic overlappings between query interfaces. Therefore, the matching should be optimized based on a particular criterion. Possible optimization criteria for matchings are *maximum number of mappings*, *maximum aggregated weight* or *stable marriage* [65].

The first criterion simply optimizes the matching such that it contains the maximum number of mappings that do not contradict the one-to-one constraint. The stable marriage criterion optimizes the global matching such that it contains only mappings where both partners have no preferable alternatives. More in detail, it locally compares each two pairs of local mappings whether both of them are *stable*. For example, consider the four elements a_1, a_2, b_1, b_2 of two schemas A and B. Suppose there are two mappings (a_1, b_1) and (a_2, b_2). Then, the algorithm investigates whether an element of the first

mapping (e.g. a_1) prefers the alternative partner (b_2) to the current mapping partner (b_1). A mapping configuration is called stable if no such alternative preference exists among all pairs of mappings. In contrast, the maximum weight criterion computes a global matching that has the best aggregated weight value. We follow the maximum aggregated weight paradigm because in our setting it reflects best the desired optimization criterion of maximum semantic overlap among the interfaces. We express our specific weights in similarity scores. We define:

Definition 4.2 (Optimal Interface Matching) *Let M_{all} be the set of all valid matchings between two query interfaces and $M \in M_{all}$. Let further Sim be a measure for the quality of a particular mapping between query interface elements. M is called optimal if*

$$\nexists M' \in M_{all} : \sum_{m'_i \in M'} Sim(m'_i) > \sum_{m_i \in M} Sim(m_i)$$

Having introduced the matching method in general, we describe each step of the method in detail in the next sections.

4.2 Construction of a Domain Dictionary

As emphasized in Chapter 3, there is a parallel between natural language texts and Human-oriented query interfaces. The terms of a text are an important source to discover the semantics of the text. Similarly, the terms appearing on a query interface describe its semantics and therefore they may be used for the identification of mappings among the elements of the query interface [91].

It is difficult to identify mappings between elements when relying only on their labels as they appear on the Web page. The main reason is that designers choose rather phrases than single identifying nouns for the element labels. This leads to heterogeneous flections and stopwords. Also, terms in labels may be used homonymously or, a single concept may be labeled by multiple but synonym terms on different interfaces. Therefore, the comparison of labels requires a preprocessing step. We accomplish this preprocessing step by constructing a *dictionary* of terms from a set of interfaces in a particular domain.

We obtain the elements of the dictionary from all label texts that are shown on a query interface by utilizing several normalization steps. First, we identify and remove domain specific *stopwords*. Then, we normalize all remaining terms by stemming and identifying their lemmas. Finally, synonymous forms among the remaining terms are looked up and grouped together. These synonym groups form the dictionary of a query interface or a set of query interfaces in a given application domain.

We define the dictionary and its elements as follows:

Definition 4.3 (Synset) *A synset S is a set of terms such that all terms in a synset are synonym to each other:*

Definition 4.4 (Dictionary) *A dictionary D for a given domain is a set of synsets that all describe real world objects in D.*

4.2.1 Identification of Content Words

A label may contain two classes of words: *content words* and *stopwords*. In contrast to content words stopwords do not carry domain specific semantics. They are necessary to build grammatically correct phrases or sentences in natural language. The proper identification and removal of stopwords is an important issue for integration purposes [23] because stopwords may lead to improper mappings. In the example phrase `Enter the airport code` that may appear as a label of a field on a query interface, the word `airport` is a content word whereas the word `the` is a stopword. The further word describes the concept of the field whereas the latter does not give any hint to the semantics of a the field. General purpose stopword lists have been developed for classical natural language processing. They are available online[1].

In contrast to texts, query interfaces use a very restricted vocabulary. Labels usually contain only single terms or short phrases. Particular terms may carry specific semantics in the context of a specific application domain that differs from their general purpose meaning. A stopword in general may become a content word when used as a label in a query interface of a particular domain. For example, the term `to` is a stopword in common understanding, but in the airline domain, it might be used as standalone label for a field denoting the destination of a flight. Therefore, a domain specific stopword handling is mandatory when integrating query interfaces.

We use ideas that originate in an adjacent project [23] and adjust them for our purposes. We obtain the set of domain specific content words CW from the the set of all terms T by filtering T with a commonsense stopword list and additionally checking the following two assertions for each term $t \in T$.

(A1) No Label Assertion: A label cannot be empty.

(A2) Sibling Assertion: All sibling labels of an interface are distinct. Moreover, they cannot be in a hyper-/hypo- or synonymy relationship.

First, we apply (A1). We check each label whether it becomes empty after stopword removal. If this case applies, the causing term is considered as a content word and removed from the stopword list. For example, if an element has only a label `to`, the commonsense stopword `to` becomes a content word for this application domain and is not considered for the domain specific stopword list.

Next, we apply (A2). If two initially sibling labels of a single interface equal after stopword removal we consider the causing commonsense stopword as content word. An example are the two labels `from city` and `to city`. A common stopword list includes

[1] We utilize the stopword list from the onix text retrieval toolkit, see http://www.lextek.com/manuals/onix/stopwords1.html for more information.

both `from` and `to`. Stopword removal with a commonsense stopword list would lead to the removal of both terms from the vocabulary. The remaining label of both fields would be `city`. In this case, both fields could not be distinguished anymore using their labels. Therefore, the terms `from` and `to` have to be considered as content words.

Algorithm for Content Word Computation

Algorithm `computeContentWords` (see Algorithm 4.1) estimates the domain specific set of content words for a set of schema trees ST, the set T of all terms that appear in the schema trees and a domain-independent stopword list SW. It results the set CW of content words. First, the algorithm initializes CW with T. It next removes all stopwords (SW) from the initial set of content words. It iterates over all elements of SW and checks for each node in every schema tree the two assertions as described above.

Algorithm 4.1 ComputeContentWords(ST, T, SW)

Input: a set of schematrees ST, the set of terms T and a list of common sense stopwords SW
Output: set of content words CW

```
CW = T \ SW;
for all sw ∈ SW do
  for all node ∈ ST do
    reductLabel = Label(node)\sw;
    if (reductLabel = ∅) then
      CW := CW ∪ sw;
    else
      for all s ∈ Siblings(n) do
        reductSiblLabel = Label(s) \sw;
        if reductLabel ⊆ reductSiblLabel ∨ reductSiblLabel ⊆ reductLabel then
          CW := CW ∪ sw;
        end if
      end for
    end if
  end for
end for
return CW;
```

First, it checks the No Label Assertion (A1). The algorithm computes a reduced form of the current node label (*reductLabel*) by removing the current stopword from the label. Please note, that we represent a label as a set of its terms. If the reduced label equals the empty set the current stopword cannot be treated as a stopword in the domain of interest. The algorithm adds it to the set of content words CW and continues with the next entry of the commonsense stopword list.

If the current stopword is not identified as a content word by (A1) the algorithm continues with checking the Sibling Assertion (A2). Here, it first determines the sibling

nodes of the current schema tree node *node*. It computes a reduced label for each sibling node similarly to the reduced label of the current node. Next, the algorithm compares the reduced label of each sibling with the reduced label of the current node. Assertion (A2) states that sibling labels cannot be equal nor they can be in a hyponym/hypernym relationship. We utilize the observation that a hyponym of a label can be expressed by adding additional terms to a label. For example, `Departure Date` is more specific than the label `Date`.

Algorithm 4.1 implements this observation by computing subset relations of sibling labels. Therefore, the algorithm compares the term sets of the labels of the sibling node and those of the current node. If it identifies equality or a subset relationship the current stopword is added to the set of content words CW.

We also investigated an additional utilization of WordNet [30] for the advanced detection of hyponyms and hypernyms. The application cases in our concrete examples were rather minute, therefore we could not identify a significant improvement of the experimental results. Especially, we noted in some cases that the actual meanings of particular terms in the investigated domains were more specific than their general meanings, therefore hyponym/hypernmym suggestions of WordNet went wrong.

4.2.2 Stemming

In most cases, all words that originate from the same stem carry similar semantics. Therefore, nodes whose labels only differ by flections of their terms should be mapped together because they represent the same concept. Without stemming a correct mapping may fail due to the different spelling. We use the Porter Stemmer [71] to normalize words and to remove flections. For example `depart` and `departing` are both stemmed to `depart`.

WordNet utilizes a more sophisticated approach, the lemma identification (see [30]). In contrast to the stemming approach that results in artificial terms, the WordNet approach results in linguistically correct normal forms. We alternatively employed the word identification from WordNet to normalize content words, but the experiments did not show a better performance. The reason is that the differentiation of WordNet is finer than that of the stemmer and thus it fails to identify several cases.

4.2.3 Synonyms

Query interfaces may use different but synonymous words in their labels to denote semantically identical concepts. Therefore, we pairwise check all content words in the vocabulary for synonymy. We identify synonym words by utilizing WordNet. The algorithm puts synonym words together into *synsets*. For example, `depart, go, going, last, leaving` are identified as a single synset. The final dictionary is formed from the set of all these synsets.

4.2.4 Aggregation

The overall algorithm for the dictionary construction is a sequential execution of content word estimation, stemming and grouping of synsets (see Algorithm 4.2).

Algorithm 4.2 DictConstructor(ST, T, SW)

Input: a set of schematrees ST, a set of terms T, a list of stop words SW
Output: dictionary of the Set of interfaces $Dict$

CW = computeContentWords(ST, T, SW);
$stCT$ = stem(CW);
$Dict$ = groupSyns($stCW$);

4.3 Definition of Element Similarities

Each element of a query interface provides certain properties such as a label, values or information about its relative position in the interface. These properties allow to compare the element with elements from other query interfaces. We compute pairwise similarity measures based on these properties to identify those elements from different interfaces that denote identical concepts (for example, fields that denote the departure airport on different airline interfaces).

Number and kind of properties of elements differ among the interfaces. Most of the properties are not mandatory. For instance, a field may not have a label and obviously, it has no child elements. In contrast, a group of fields never has any assigned value or HTML name. Because of this heterogeneity we define several similarity measures that each emphasize a different aspect. We compute the final similarity score of two elements by combining a number of similarity measures. We distinguish between *local* similarity measures and *structural* similarity measures. Whereas the further refer to properties that can be obtained by the isolated investigation of a single element, the latter investigate the relationships of multiple elements in the interface (such as children-relationships between nodes in the interface tree). Please note, that we interchangeably use the terms *node* and *element* in this section.

The overall similarity ($NodeSim$) is computed by a weighted aggregation of the local similarity and all structural similarities. We now describe the intuition for and the computation of all similarity measures in detail.

4.3.1 Local Similarities

Local similarities utilize properties of a single element such as the label, the name and its values independently of their position in the tree. We use the following three local measures: *label similarity*, *value similarity* and *name similarity*. We define the local similarities as follows.

Label Similarity

The label is the main clue for a human user to recognize the meaning of an element, therefore the label is a promising source to unveil the semantics of a particular element of a query interface as long as the element provides a label. Similarly to the human way of recognition that assumes that a particular label term leads to a specific meaning of that label, the label based similarity assumes that syntactically similar labels lead to semantically similar elements. Therefore, it is promising when matching query interfaces that provide distinctive labels.

We compute the label similarity between two elements by following ideas from information retrieval. We use term vectors that represent labels and compute a cosine similarity score between these vectors. We prefer this more complex model (compared to a direct term comparison) because most labels contain more than a single term.

We transform label phrases into a normalized format to enable a comparison. First, we decompose labels into their terms. We call these terms *label terms*. We normalize all label terms using stemming. Next, we check for each label term whether it represents a content word of the current domain. Only label terms that represent content words remain in the label, stopwords are removed (compare Section 4.2).

We estimate the label similarity of two interface elements based on the cosine similarity known from information retrieval (compare for example [2]). Our algorithm borrows ideas from [91], but additionally uses synsets and domain specific stopwords.

Definition 4.5 (Label Vector) *Let n be a node of a schema tree, let L be its set of terms, and let T, $|T| = t$, be the set of all terms. We define*

$$LabelVec(n) = \begin{pmatrix} v_1 \\ ... \\ v_t \end{pmatrix}, \text{ where } v_i = \begin{cases} 1 & L \cap dictSynSet(i) \neq \emptyset \\ 0 & else \end{cases}$$

as the Label Vector of n.

For example, consider a domain dictionary of three synsets `{{From,Depart},{To, Arrive}, {Date}}` and the label phrase `From`. The corresponding label vector is $\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$.

We define the label similarity of two elements as the cosine similarity of their label vectors:

Definition 4.6 (Label Similarity) *Let s and t be two schema tree nodes. Then, the label similarity between s and t is*

$$LabelSim(s,t) = \frac{LabelVec(s) \cdot LabelVec(t)}{\|LabelVec(s)\| \|LabelVec(t)\|}$$

Consider again the dictionary `{{From, Depart},{To, Arrive}, {Date}}`. Suppose the two labels `From` having label vector $\begin{pmatrix}1\\0\\0\end{pmatrix}$ and `Depart` having label vector $\begin{pmatrix}1\\0\\0\end{pmatrix}$, respectively. Their label similarity is 1.

Value Similarity

Predefined values may guide human interpreters to guess the correct meaning of a field, especially for fields that allow only a fixed set of values and no label is provided. Consequently, the value similarity is best applicable for these cases. A typical example is a selection list that denotes months of the year.

The application of the value similarity is twofold. On the one hand, value sets provide a good criterion to distinguish concepts that do not overlap semantically. For example, a field containing month names can easily be distinguished from a field containing numbers of days. On the other hand, the value similarity may lead to ambiguities when comparing fields that represent semantically close concepts. For example, in the airline domain the value similarity may not be able to distinguish a field for the departure date from a field denoting the return date because both fields may contain similar value sets.

We now define the computation of the value similarity. HTML encodes all values in a string data type. Therefore, we utilize the string edit distance to estimate the value similarity. We first define a measure for the similarity of a single pair of values. This measure is based on the Levensthein distance of strings (string edit distance). Let be s_1 and s_2 two strings, then the string edit distance $StrEditDist(s_1, s_2)$ estimates the number of single character operations that have to be executed to transform s_1 into s_2 or s_2 into s_1, respectively [54]. The maximum number of operations possible equals the length of the longer string. Therefore, we normalize the Levensthein distance measure by dividing it by the length of the longer string.

We define the similarity of two single values v and w ($ValPairSim$) as follows:

Definition 4.7 (Value Pair Similarity) *Let v and w be two values of different elements, then the value pair similarity between v and w is*

$$ValPairSim(v, w) = 1 - \frac{StrEditDist(v, w)}{max(|v|, |w|)}$$

We define the value similarity $ValueSim$ utilizing the value pair similarity:

Definition 4.8 (Value Similarity) *Let s and t be interface nodes having value sets $Val_s = \{val_{s1}...val_{sn}\}$ and $Val_t = \{val_{t1}...val_{tm}\}$. Then, the value similarity between s and t is*

$$ValueSim(s, t) = \frac{\sum_{val_s \in Val_s} \sum_{val_t \in Val_t} ValuePairSim(val_s, val_t)}{|Val_s||Val_t|}$$

The value similarity $ValueSim$ simply averages the pair-wise similarities between all possible value pairs to obtain a measure of the similarity between the value sets of two interface nodes. We use a complete pair-wise comparison because this approach assures a general purpose solution. For example, an alternative order-aware comparison would require a similar ordering of both value sets. Also, it requires additional adjustments in the case that both values sets differ in their size.

We make two additional adjustments to the value similarity. First, if both interface tree nodes do not provide values, we set the value similarity to *null*. Second, if only one interface tree node provides a set of values, we set the value similarity to 0. In the further case the value similarity cannot be applied to get any conclusion. In contrast, in the latter case it is clear that both compared elements differ in terms of their values.

Name Similarity

Developers of query interfaces define names for the fields in order to assign the transfered values to the correct parameters of an HTTP request. We assume, that fields correspond to database objects such as attributes. Therefore, the (internal) HTML field names may provide an intensional view to the names of the database attributes. Under this assumption, it is useful to compute a name based similarity between fields.

Names on query interfaces are used similarly to variables in programming languages. There is no common policy on naming fields. Nevertheless, in many of the query interfaces the names to some degree reflect the semantics of the named field. Whereas fields always provide a name this property is not available for groups because they are not directly reflected in the HTML source.

There are syntactic restrictions on names that have to be covered in the similarity measure. For example, names of fields cannot contain spaces. Also, it is a common practice to abbreviate the terms in a field name and to combine multiple abbreviated terms to a single name. The reason for this practice is that early generation programming languages did not accept variable names that exceeded a certain length.

We perform several preprocessing steps before computing the name similarity. We first clean non-alpha characters such as brackets or numbers. Next, we identify delimiters in the name. Common delimiters include upper-case letters, underscores and slashes. We then split the name at the position of the delimiters into terms. For example, a name of a particular field may be `DepFrom`. After splitting we obtain `{dep, from}`.

We suppose that the dictionary contains the relevant terms encoded in field names. We look up all obtained terms in the dictionary. Because of the abbreviation problem (see above) we do not enforce full equality between a dictionary entry and the term, but concentrate on dictionary entries that start with the same letters than the current term of interest. Terms that can be identified in the domain dictionary are substituted by the dictionary entry that best matches it. If no match can be identified the term is dropped.

Assume, for example, a dictionary contains the three synsets `{{From,Depart},{To, Arrive}, {Date}}` and a field is named `DepForm`. We match the term `dep` to the dictionary term `depart`. We also find `from` in the dictionary. We achieve a set-like field name representation similarly to the label representation introduced in Section 4.2.

We transform the set representation of a name into a vector $NameVec$ to compute the similarity between names such that all those components v_i in $NameVec$ are set to 1 where the name contains a member of the $i.$ synset in the dictionary (function $dictSynSet(i)$) , all other components of $NameVect$ are set to 0:

Definition 4.9 (Name Vector) *Let n be a node of the schema tree. The name vector of n is*

$$NameVec(n) = \begin{pmatrix} v_1 \\ ... \\ v_n \end{pmatrix}, where\ v_i = \begin{cases} 1 & Name(n) \cap dictSynSet(i) \neq \emptyset \\ 0 & else \end{cases}$$

We than define the name similarity between two nodes of the schema tree by the cosine similarity of their name vectors:

Definition 4.10 (Name Similarity) *Let s and t be two schema tree nodes, then the name similarity between s and t is*

$$NameSim(s,t) = \frac{NameVec(s) \cdot NameVec(t)}{\|NameVec(s)\|\|NameVec(t)\|}$$

We compute the name similarity only for leaf nodes. In contrast to leaf nodes internal nodes do not have an HTML name. Therefore, we set the name similarity for internal nodes to `null`.

Aggregation

We aggregate the three previously introduced local similarities to a single score, named *Local Similarity*. More in detail, the local similarity is a weighted aggregation of the label similarity $LabelSim$, the name similarity $NameSim$ and the value similarity $ValueSim$.

The formal definition of the local similarity is as follows:

Definition 4.11 (Local Similarity) *Let s and t be two schema tree nodes and w_{LL}, w_{LN} and w_{LV} be weights. Then the local similarity between s and t is*

$$LocSim(s,t) = \frac{w_{LL}LabelSim(s,t) + w_{LN}NameSim(s,t) + w_{LV}ValueSim(s,t)}{w_{LL} + w_{LN} + w_{LV}}$$

The name similarity and the value similarity may not be obtained for all nodes. Therefore, they may be set to `null`. In this case, we set their corresponding weights to 0. The weights w_LL, w_LN and w_LV allow domain dependent adjustments. In our experiments we learn them using separated learning sets (compare Section 4.5).

4.3.2 Structural Similarities

In contrast to local similarity measures, structural similarity measures consider the tree structure that surrounds particular nodes. We introduce the three structural similarity measures *sibling similarity*, *child similarity* and *order similarity* in the following.

Sibling Similarity

Query interface elements that have similar neighbors (siblings in the schema tree) often are semantically close. Designer of query interfaces express this relationship by a visual grouping of the elements. For example, a field `departure day` often is grouped together with a (sibling) field `departure month` and/or a field `departure year`. The sibling similarity takes this observation into account and computes a score between two elements by aggregating the local similarities of their sibling elements. We define the sibling similarity between two nodes as follows.

Definition 4.12 (Sibling Similarity) *Let s and t be two nodes of a schema tree and let $SBL_s = \{sbl_{s1}...sbl_{sn}\}$, $SBL_t = \{sbl_{t1}...sbl_{tn}\}$ be the set of their sibling nodes. The sibling similarity between s and t is*

$$SiblSim(s,t) = \begin{cases} null & \text{if } SBL_s = \emptyset \vee SBL_t = \emptyset \\ \frac{\sum_{sbl_s \in SBL_s} \sum_{sbl_t \in SBL_t} LocSim(sbl_s, sbl_t)}{|SBL_s||SBL_t|} & else \end{cases}$$

Child Similarity

Humans derive the meaning of particular elements also by considering their child or parent elements. On a query interface, the parent-children relationship is visually expressed by a group of aligned elements that share a single label. For example, a group labeled `Departure Date` frequently is composed of three fields `departure day`, `departure month` and `departure time`. We hypothese, that the children of a particular node s have a high impact on its semantics. Nodes that share similar children are likely to be semantically close. This is reflected in the child similarity. It is a measure based on a comparison of child elements of (internal) schema tree nodes. We formally define:

Definition 4.13 (Child Similarity) *Let s and t be nodes in the schema tree and let $CLD_s = \{cld_{s1}...cld_{sn}\}$, $CLD_t = \{cld_{t1}...cld_{tn}\}$ be the sets of their direct child nodes. Then the child similarity between s and t is*

$$ChildSim(s,t) = \begin{cases} null & \text{if } CLD_s = \emptyset \vee CLD_t = \emptyset \\ \frac{\sum_{cld_s \in CLD_s} \sum_{cld_t \in CLD_t} LocalSim(cld_s, cld_t)}{|CLD_s||CLD_t|} & else \end{cases}$$

Order Similarity

We observed that most query interfaces of a particular domain follow a common ordering of their elements. We transform this observation into the assumption that the variability of the relative position of a particular element on a query interface inside a particular application domain is minimal.

We already exploit the order information during the extraction phase (compare Section 3.2.2). Therefore, the order of the nodes in the schema tree reflects the element order intended by the designer of the query interface.

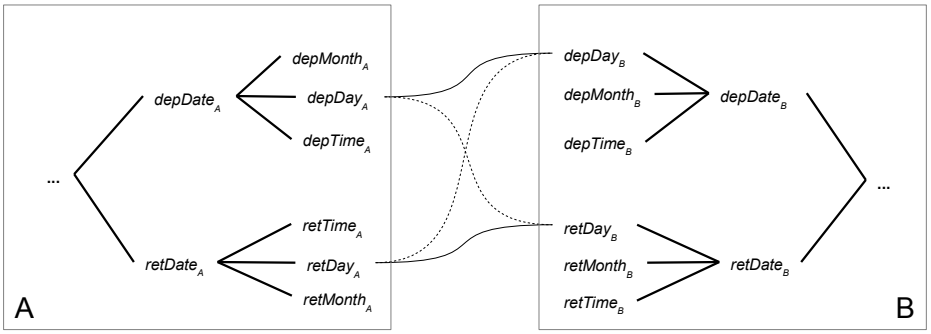


Figure 4.3: Example order conflicting mappings.

For example, Figure 4.3 shows again the two snippets of the interfaces A and B in our motivating example (compare Figure 1.4 on Page 7). Both interfaces contain fields for departure day information (fields $depDay_A$ and $depDay_B$) as well as return day information (fields $retDay_A$ and $retDay_B$). When considering only local properties of the mapped nodes, it is likely that the matching algorithm computes the same similarity for all of the four candidate mappings ($depDay_A$, $depDay_B$), ($depDay_A$, $retDay_B$), ($retDay_A$, $depDay_B$) and ($retDay_A$, $depDay_B$) that form two possible matchings. Although all mappings receive the same similarity, the preferable matching is clearly {($depDay_A$, $depDay_B$), ($retDay_A$, $retDay_B$)} (shown as solid lines). The other possible configurations {($depDay_A$, $retDay_B$),(ret_ADay, $depDay_B$)} (shown as dotted lines) do not reflect the semantic correspondences.

When comparing both interfaces, we observe that both of them request the departure day information *before* the return day information. The differences of the two possible matchings are the relative positions of the elements in the schema trees. The relative position of the departure date information as well as the return date information in both schema trees equals.

The order similarity reflects this assumption. It peaks if the relative position of the mapped interface elements is similar in both respective schema trees. We define the order similarity to reflect this property:

Definition 4.14 (Order Similarity) *Let S and T be two schema trees and elements $s \in S$ and $t \in T$ be nodes of them and let the function $postOrderId$ return the number of*

the tree node when traversing postorder. Then the order similarity OrdSim is defined:

$$OrdSim(s,t) = 1 - abs(\frac{1}{|S|}postOrderId(s) - \frac{1}{|T|}postOrderId(t))$$

The order similarity is computed as inverse normalized distance between the postorder node ids of the two nodes in a mapping. We use the postorder traversal because it better reflects the initial interface field order than a preorder traversal. The reason is, that the initial order as defined by the interface designer covers only the leaves of the tree. However, we also introduced internal nodes during interface extraction. The postorder traversal minimizes their influence compared to preorder traversal.

4.3.3 Overall Similarity

The overall similarity is aggregated from the local similarity and the structural similarities *SiblSim*, *ChildSim* and *OrdSim*. It reflects all aspects that express particular semantics in a certain interface.

We show in Section 4.5 that this ensures a better precision and recall compared to a restricted similarity estimation based on only terms or tree structure. The weights w_L, w_S, w_C and w_O for each of the component similarities allow adjustments to particular domains or datasets. In our experiment, we learn them from separated learning sets (compare Section 4.5).

We define the overall similarity as follows:

Definition 4.15 (Node Similarity) *Let s and t be nodes in the schema tree and w_L, w_S, w_C and w_O be weights. Then the node similarity between s and t is*

$$Sim(s,t) = \frac{w_L LocSim(s,t) + w_S SiblSim(s,t) + w_C ChildSim(s,t) + w_O OrdSim(s,t)}{w_L + w_S + w_{Child} + w_O}$$

4.4 Global Matching

Having defined local and structural similarity measures for the computation of element mapping candidates we now describe the computation of a global matching between two query interfaces from these element mappings. More precisely, a global matching is a set of non-adjacent element mappings that (hopefully) best reflects the semantic overlappings between the two interfaces (see Definition 4.2). We use the terms global matching and optimal matching synonymously. We describe two different methods to compute a global interface matching for a set of query interface element mappings.

4.4.1 Hungarian Method

The Hungarian Method [47] is a global optimization technique, that was developed for the problem of finding a global cost minimum when assigning workers to tasks. This problem is known as *assignment problem* [13]. We adopt our problem of finding a global maximum to the Hungarian Method by transforming our maximization problem into a related problem for searching a global minimum. This transformation is straightforward and can be achieved by substituting the similarity scores of all mapping candidates by a distance score. We obtain the distance score for a particular mapping by subtracting its similarity score from the maximum similarity among all mappings.

The Hungarian Method is built up from the bigraph understanding of the matching problem (compare Section 4.1). It represents the bigraph by its adjacency matrix. A matching in the adjacency matrix is always a set of matrix elements that do not share columns or rows. The best matching is the maximum set of such elements with a minimal value. The algorithm manipulates rows and columns of the adjacency matrix such that it finally obtains a maximal matrix element set of this style where all values are zero. For further insight into the Hungarian Method please refer to [47].

4.4.2 Greedy Approach

We present an alternative *greedy* approach to the Hungarian Method that allows us to compare different optimization strategies and to draw parallels to the related work. The greedy approach investigates and resolves *mapping conflicts*. A mapping conflict occurs whenever a particular node has candidate mappings with more than one node of a particular interface. In contrast to the Hungarian Method, the conflict resolution approach optimizes the mappings only locally. Beside of the mapping candidate similarity scores it exploits an additional order predicate that cannot be transformed into a similarity score and therefore it is not applicable in the Hungarian Method. This idea is influenced by [91] and adopted to our problem. We also use this approach to make our results comparable to the related work.

We define a conflict-free mapping:

Definition 4.16 (Conflict-free Mapping) *Let* $S = \{s_1, ...s_n\}$, $T = \{t_1, ..., t_m\}$ *be query interfaces,* M *be the set of mappings between* S *and* T. M *is said to be conflict-free iff:*

$$(i) \forall s \in S, t_1, t_2 \in T : (s, t_1) \in M \wedge (s, t_2) \in M \rightarrow t_1 = t_2$$

$$(ii) \forall s_1, s_2 \in S, t \in T : (s_1, t) \in M \wedge (s_2, t) \in M \rightarrow s_1 = s_2$$

Algorithm 4.3 describes the greedy conflict resolution in detail. It iterates over all mapping candidates m and looks for mapping conflicts. A configuration of mappings and its conflicting *alternative mappings* form mapping conflicts. Therefore, the algorithm identifies conflicting mappings by looking for their alternative mappings. We distinguish alternative source mappings and alternative target mappings.

Algorithm 4.3 ComputeGreedyMatching(S, T, M)

Input: two query interfaces S and T, set of mapping candidates M
Output: conflict free set of mappings M

while M changes **do**
  for all $s \in S, t \in T$ **do**
    m := (s,t);
    $m_{as} := (s, t_a) : t_a \in T \wedge t_a \neq t \wedge \nexists t_i \in T : t_i \neq t_a \wedge sim(s, t_i) > sim(s, t_a)$
    $m_{at} := (s_a, t) : s_a \in S \wedge s_a \neq s \wedge \nexists s_i \in S : s_i \neq s_a \wedge sim(s_i, t) > sim(s, t_a)$
    if $(m_{as} \neq null \wedge m_{at} \neq null) \wedge (order(s, s_a) = order(t, t_a)$ **then**
      $M := M - m$
    else
      if $(m_{as} \neq null \wedge m_{at} = null) \wedge (sim(m) \leq sim(m_{as}))$ **then**
        $M := M - m$
      else
        if $(m_{as} = null \wedge m_{at} \neq null) \wedge (sim(m) \leq sim(m_{at}))$ **then**
          $M := M - m$
        end if
      end if
    end if
  end for
end while
return (M);

We define them as follows:

Definition 4.17 (Alternative Mappings) *Let S, T be query interfaces, M be the set of mappings between S and T, Let further be $m \in M$ and $m = (s, t)$.*
A mapping $m_{as} = (s, u)$ is called alternative source mapping to m iff $u \neq t$.
A mapping $m_{at} = (v, t)$ is called alternative target mapping to m iff $v \neq s$.

If alternative mappings exist for a particular mapping candidate m, the algorithm identifies the *best* alternative mapping candidate. The best alternative mapping is defined as the one that has the highest similarity score among all alternative mappings. We distinguish tree cases of conflicts: (1) alternative mappings for both source as well as target elements of m exist, (2) only the source element of m has alternative mappings and (3) only the target element of m has alternative mappings. The algorithm handles all three cases. In Case (1), the algorithm computes a configuration of mappings that uses the best alternative mappings and avoids the current mapping m. If these alternative configuration fulfills the order predicate, the current mapping m is dropped from the set of candidate mappings M. In Cases (2) and (3), the algorithm compares the similarity scores of m and its best alternative mapping. If the current mapping m has a lower or equal similarity score than the alternative mapping, it is dropped from M.

4.4.3 Comparison

The main advantage of the Hungarian method is its ability to optimize globally. In contrast, the greedy method in some cases fails to find the optimal solution, because it uses only local knowledge while proceeding. Its strength depends on the preprocessing effort. More in detail, it requires an additional threshold that differentiates mapping candidates from non-mapping pairs. On the other hand, the greedy method allows to introduce the optimization criterion of order conflict-freeness (see Definition 4.16) The Hungarian Method cannot include this criterion because it relies on binary predicates that can be expressed in similarity scores. The order-conflict criterion compares at least three mapping candidates to distinguish a conflicting from a conflict free configuration.

Both Methods have the complexity $O(n^3)$, n refers to the number of elements in the larger one of the two interfaces to be matched. The complexity of the greedy approach is caused by the three nested loops that iterate over the mapping candidate set (see Algorithm 4.3). For the details of the Hungarian Method please see [47]. However, when using the greedy method, the size of the input candidate mapping set can be significantly reduced. A preprocessing step may prune candidates by defining a lower threshold for similarity scores. Only candidate mappings that meet the threshold will be considered for the optimization. We figured out experimentally that the pruning does not significantly influences the results of the greedy method while improving its performance (data not shown). The pruning is not useful for the Hungarian Method because a globally optimized matching may contain mappings with low similarity whereas a local

optimizer (such as our greedy method) does not consider such mappings in general. In the experimental section we will show that the Hungarian Method outperforms the greedy approach in our setting (compare Section 4.5).

4.4.4 Complete Algorithm

The overall matching algorithm brings together all steps described before. It takes as input a set of query interfaces ST in their schema tree representation. It outputs a set of node mappings $M+$ that represents the global matchings of all pairs of query interfaces that can be constructed from ST. First, the algorithm constructs the candidate mappings by generating pairs of nodes (s, t). Note, that in these pairs s always belongs to another interface than t. Inverse pairs (t, s) are omitted due to the reflexivity property of the mapping relation. The algorithm computes similarity scores for all pairs (s, t) as described above using the function `ComputeSimilarities`. This function is straightforward and therefore omitted here.

Algorithm 4.4 ComputeMappings(ST)

Input: a set of schematrees ST
Output: Set of mappings M

```
N := {n|n ∈ Nodes(ST)};
M := ∅
for all S,T ∈ ST, Id(S) < Id(T) do
  for all s ∈ S, t ∈ T do
    m := (s,t);
    M := M ∪ m;
  end for
end for
ComputeSimilarities(M);
M+ := ∅;
for all S,T ∈ ST,Id(S) < Id(T) do
  M+ := M+∪ ComputeGlobalMatching(S, T, M);
end for
return M+;
```

The next step is the computation of the global matchings. In the algorithm, we put the proxy function `ComputeGlobalMatching` for this purpose. This function can either be instantiated by the Hungarian Matcher (compare Section 4.4.1) or by the Greedy Matcher (compare Section 4.4.2). The function computes a global matching for a single pair of interfaces. The final set of all element mappings $M+$ among multiple query interfaces is obtained by the union of the global matchings for all pairs of query interfaces to be matched.

4.5 Evaluation

We conducted a large empirical evaluation of our matching approach. The results are presented in this section. First, we give information about the used dataset in Section 4.5.1. Then, we evaluate the quality of all similarity measures in Section 4.5.2. Finally, we apply our algorithm to the interfaces of the gold standard to obtain a measure for the quality of our matching method in Section 4.5.4.

4.5.1 Datasets

In this section, we describe structure and construction of our mapping gold standard.

Structure

We used 150 interfaces from 7 domains for the experimental evaluation of the matching algorithm. The domains are airline, auto, book, job, real estate, hotel and carrental. The interfaces originate from the enriched ICQ-dataset as collected for [25]. Table 4.1 gives more details about the mappings of the schema tree nodes of these interfaces. Except for the hotel domain, where the gold standard contains 30 interfaces, all other domains consist of 20 interfaces.

The schema tree representations of the 150 interfaces in the gold standard together contain 1727 nodes. Among the seven domains the auto (-sale) domain has the smallest number of schema tree nodes on average. The 20 interfaces of this domain contain only 147 schema tree nodes. The airline domain contains the most complex interfaces on average. The 20 interfaces of this domain contain 382 schema tree nodes. This observation underlines the domain specific design of interfaces. While in the auto domain a few fields are sufficient to get all information, in the airline domain there are multiple fields mandatory to define a precise query.

Table 4.1 shows also the number of element pairs between interfaces of a single domain. This number is the upper boundary for the number of mappings. We use the set of pairs as initial candidate set for the computation of the mappings. This number ranges between 9814 in the auto domain and 47457 in the airline domain.

The last column of Table 4.1 denotes the number of correct mappings according to the gold standard. We observe, that the number of mappings does not correlate with the number of nodes in the particular domains. For example, in the job domain we obtain 166 nodes and 504 mappings whereas in the real estate domain we obtain 243 schema tree nodes and also 504 gold standard mappings. The reason for this behavior is the heterogeneous structure of the domains. Whereas in some domains the overall set of possible concepts is small and interfaces therefore overlap to a high degree, other domains contain highly heterogeneous interfaces that overlap only partially.

We also group the elements of the query interfaces by their semantic concepts. Table 4.2 shows the distributions of concept occurrences. Large numbers mean that in a particular domain most interfaces overlap semantically. This indicates that there is a common understanding about the semantics of a domain. In the airline, job, carrental and hotel

Domain	Inter-faces	Nodes	Nodes/Interface (Avg)	(StdDev)	Depth (Avg)	Pairs	GS Mappings
Airline	20	382	19.1	7.7	2.5	47457	1352
Auto	20	147	7.34	4.0	1.45	9814	494
Book	20	188	9.4	6.34	1.5	16388	567
Job	20	166	8.3	4.55	1.55	12883	504
Real Est.	20	243	12.5	6.99	1.95	27334	504
Carrental	20	310	15.5	4.60	2.5	45438	1289
Hotel	30	339	11.3	4.14	2.3	28622	1202
All	150	1727	11.9	5.48	1.96	187936	5912

Table 4.1: Characteristics of the mapping gold standard.

domain exists at least one concept that all interfaces share. Although interfaces in the auto and real estate domain miss such ubiquitous concepts, there are always concepts that can be found on almost all interfaces of these domains. Out of the investigated domains the airline domain provides the highest number of semantic overlappings of their interface elements. Here, out of 17 concepts, six could be identified on most of the interfaces. In contrast, in the real estate domain only one concept out of 18 is present on most of the interfaces.

Domain	Number of dist. concepts	Number of concept clusters with size $= 20$	≤ 19	≤ 15	≤ 10	$= 2$
Airline	17	3	3	3	4	4
Auto	14	0	2	1	9	2
Book	17	0	2	2	9	3
Job	12	1	2	0	7	2
Real Estate	18	0	1	3	12	2
Carrental	28	1	4	6	13	4
Hotel (20)	21	2	1	3	12	3

Table 4.2: Distributions of concept occurrences.

Construction of Gold Standard

We construct the gold standard such that we first investigate each domain for their concepts. Then, we define unique concept names that represent fields or groups of fields in the interfaces of the domain. For example, in the airline domain we defined the two concepts `departure date` and `departure day`. The further one is the more generic concept compared to the latter one. Usually, the concept `departure date` refers to a group of fields (internal node in the schema tree). This group contains the fields that constitute the departure date. A constituting field may be, among others, for example `departure day`. We define a mapping to one of the previously introduced concepts

for all schema trees nodes in the gold standard. Therein, only 1:1 mappings between concepts and schema tree nodes are permitted.

4.5.2 Evaluation of Similarity Measures

We separate a subset from the above described gold standard for adjustments of thresholds. This dataset contains 10 interfaces from the hotel domain. Please note, that the overall dataset in the hotel domain contains 30 interfaces. We put 10 of them in a separate subset for special investigations. The remaining 20 are part of the test set. The interfaces of the latter subset contain 122 query interface elements. The elements form 6584 mapping candidates. Out of them 260 mapping candidates are true positives.

We first investigate the impact of the different similarity measures as introduced in Section 4.3. A good similarity measure should be able to separate true positives from the false ones. Therefore, we quantify the impact of each measure by comparing the relative number of mappings in the overall dataset to the relative number of correct mappings for particular value ranges of their similarity scores. Figures 4.4, 4.5 and 4.6 show different distributions of values for 10 value ranges between 0 and 1. Figure 4.4 shows the distribution for the local measures *LabelSim*, *ValueSim* and *NameSim*, Figure 4.5 shows the distribution of the structural similarities *ChildSim*, *SiblSim* and *OrderSim*. Figure 4.6 concentrates on the aggregated similarities *LocalSim* (combined from all local similarities) and the overall similarity *NodeSim*.

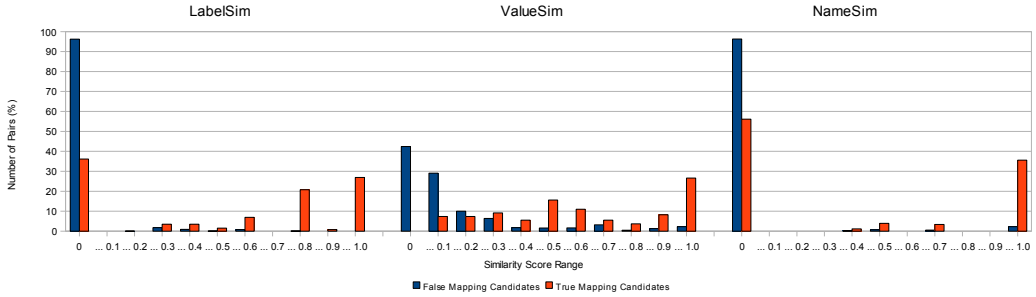


Figure 4.4: Similarity value distribution for local similarities.

Consider first Figure 4.4. The label similarity (left most diagram) shows that only 4% of the false mapping candidates obtain a label similarity higher than 0, around 1% (63 pairs) higher than 0.5. Among the true mapping candidates 64% obtain a label similarity greater than 0.2. Given a high threshold, the measure provides a good precision but the recall may be insufficient. The value similarity (middle diagram) shows a slightly different pattern. Here, an ideal threshold may be 0.3. Assuming this threshold the similarity would deliver 76% of the true mapping candidates but also 12% of the false candidates. The recall is better than that of the label similarity but the precision decreases. The name similarity is not able to identify 56% of the correct mappings, therefore, it does not provide a good recall. The precision is comparable to that of the

value similarity. Given a threshold of 0.3 the name similarity would identify 44% of the correct mappings and 3% of the false candidates.

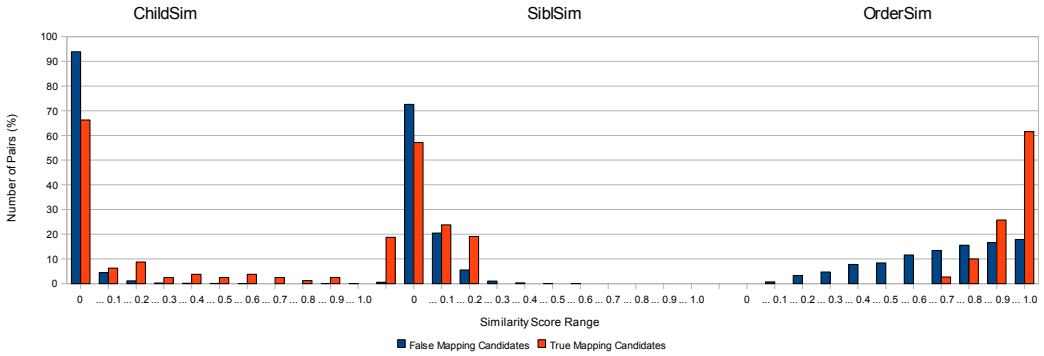


Figure 4.5: Similarity value distribution for structural similarities.

Figure 4.5 shows the value distributions for the structural similarities ChildSim, SiblSim and OrderSim. We compute the child and sibling similarity for internal node mappings. The similarity score between internal nodes often cannot be computed sufficiently by utilizing only local similarities, because internal nodes lack information such as a name or values. For the child similarity only 0.5% of the false mapping candidates achieve a score above or equal 0.3. Therefore, the precision of this measure is high. Out of the correct mapping candidates only 19% achieve a score above or equal 0.3, therefore, the recall of this measure is not promising. The sibling similarity never exceeds an assumed threshold of 0.3 for all correct mapping candidates, but also only 1.5% of the false candidates meet the threshold. For the given dataset, this similarity does not provide any impact, but this may be a specific to the chosen dataset.

The order similarity values are distributed differently, but the higher value ranges clearly show, that the order similarity is able to separate correct mappings from false mappings. Given a threshold of 0.9 we obtain 18% of the false mapping candidates but 62% of the correct candidates.

Figure 4.6 shows the value distributions for the aggregated similarities. We adjust the weights using a leave-one-out cross validation learning step. The left diagram shows the weighted aggregation of the three local similarities. We estimated that for this dataset a setting of all component weights $w_L = w_N = w_V = 1$ performs best. Given a threshold of 0.3 we would obtain 4% of the false candidate mappings, but 73% out of the correct mapping candidates. These numbers clearly lead to reasonable precision and recall.

The righternmost bars in the diagram show the overall similarity that aggregates all similarity measures. We compute the overall similarity by using the weights $w_{Loc} = 5$, $w_{Sibl} = 3$, $w_{Child} = 5$ and $w_{Ord} = 1$. These weight setting especially reflects the different value distributions among the structural similarities. The overall similarity achieves a better result than all component similarities, 83% of the correct mappings obtain a similarity score 0.3 or higher, only 6% of the false candidates fulfill this criterion.

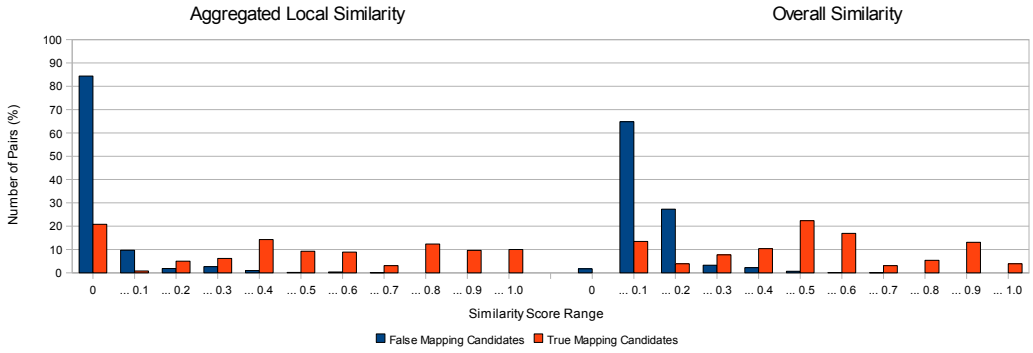


Figure 4.6: Similarity value distribution for aggregated similarities.

Summarizing, we can see that the computed similarities are of different impact and quality. The main observation is that the overall similarity seems to be able to divide positives and negatives in the data set. Also, the overall result in the choosen combination of measures outperformes the results of each single measure when applied separately.

4.5.3 Comparison of Different Methods for the Global Matching

We compare the Hungarian method as a global optimization approach (see Section 4.4.1) and the greedy approach based on mapping conflict resolution (see Section 4.4.2).

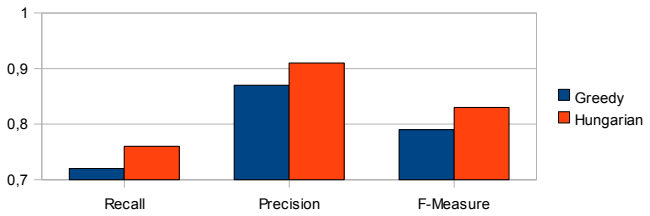


Figure 4.7: Comparison of Hungarian Method to greedy algorithm.

Figure 4.7 compares both methods in terms of recall, precision and f-measure. It shows, that the Hungarian Method outperforms the greedy approach in all three measures, in detail, the recall increases from 0.72 to 0.76, the precision from 0.87 to 0.91 and the f-measure from 0.79 to 0.83.

We also investigated the influence of additionally searching for transitive mappings as suggested by [91]. The idea of transitive mappings among query interfaces assumes that multiple ($n > 2$) interfaces are matched at once. For example, for a set of three query interfaces A, B, C having elements $a \in A$, $b \in B$ and $c \in C$ a transitive mapping (a, c) is derived from given mappings (a, b) and (b, c).

Whereas [91] reports an improvement of the matching results when exploiting transitivity, our experiments did not validate these findings. In most cases, the results of both

methods were not influenced by the computation of transitive mappings, in a few cases the accuracy even decreased. Therefore, we did not integrate transitivity computation in our final algorithm.

4.5.4 Evaluation Results

Having investigated components of the algorithm we present in this section the results of our central experiment utilizing the complete data set. We performed our matching experiment as follows. For each domain we randomly partitioned the 20 interfaces into a learning set of 15 interfaces and a test set of 5 interfaces. We learned from the learning set the optimal mapping threshold and the weights for each of the similarity measures. For each test set we estimated the recall, precision and f-measure for the mappings predicted with these parameters. We repeated the experiment 10 times using different partitions. We computed the final numbers for recall, precision and f-measure given by averaging the results of the 10 runs of the experiment. The results are shown in Figure 4.8 and Table 4.3. Whereas the figure shows only the average numbers of recall, precision and f-measure for each domain investigated, the table also reports the numbers for the best and worst results (columns `min` and `max`) and the standard deviation (column ±) among the 10 runs of each experiment.

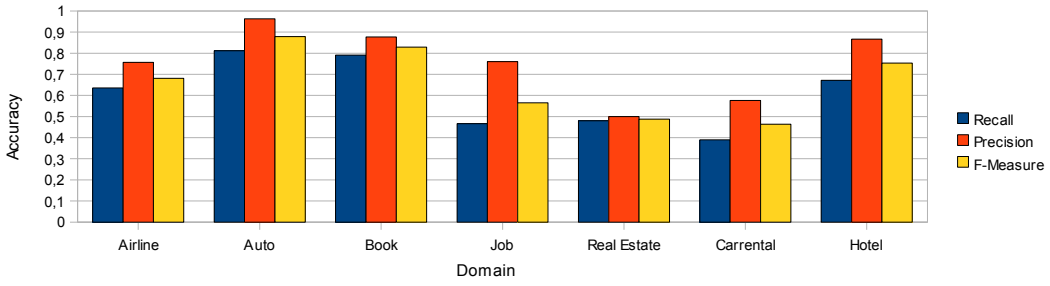


Figure 4.8: Results of the matching experiment.

Dom.	Recall				Precision				F-measure			
	avg	min	max	±	avg	min	max	±	avg	min	max	±
Air.	0.64	0.36	0.79	0.18	0.76	0.64	0.89	0.09	0.68	0.47	0.80	0.14
Auto	0.81	0.72	0.94	0.07	0.96	0.73	1.00	0.08	0.88	0.73	0.97	0.07
Book	0.79	0.69	0.90	0.08	0.88	0.79	0.96	0.06	0.83	0.75	0.92	0.05
Job	0.47	0.26	0.67	0.14	0.76	0.56	1.00	0.12	0.57	0.38	0.71	0.12
R.E.	0.48	0.25	0.69	0.12	0.50	0.38	0.69	0.09	0.49	0.30	0.69	0.10
Carr.	0.39	0.28	0.44	0.05	0.58	0.46	0.70	0.07	0.46	0.35	0.52	0.05
Hotel	0.67	0.50	0.87	0.15	0.87	0.76	0.94	0.08	0.75	0.60	0.89	0.12

Table 4.3: Absolute numbers of the mapping experiment.

We investigated different correlation assumptions. First of all, the performance does not correlate to the complexity of the interfaces. There are domains (e.g. airline or hotel) that have complex interfaces (high average number of nodes, large average depth) and the algorithm performs reasonably well. Next we investigated whether the performance correlates to the dictionary size. We figured out, that there is a slight correlation (data not shown). In domains that have a smaller dictionary the algorithm performs better than in those with a comparably bigger dictionary (compare Table 4.1). When investigating the results in Figure 4.8, we first observe that the performance depends highly on the application domain. For example, the f-measure ranges from 0.46 in the carrental domain to 0.88 in the auto domain. The average f-measure over all seven domains is 0.67. The standard deviation of the f-measure as shown in the last column of Table 4.3 shows the heterogeneity of the domains. First, there are domains in which the algorithm performs well in all experimental configurations. For example, in the book domain the standard deviation is 0.05 and the overall performance is good. In other domains such as carrental the algorithm performs bad for all configurations (f-measure=0.46, stdev=0.05). Also, there are domains that receive a good average f-measure, but the algorithm shows a heterogeneous performance on different runs. An example is the airline domain with an f-measure of 0.68 and a high value of 0.14 for the standard deviation. Consequently, the success of the algorithm in this domain depends on the given interfaces.

These examples of the results show that it is difficult to explain the results without considering domain specifics. Therefore, we discuss the results per domain.

Airline Although the airline domain contains complex query interfaces, the matching algorithm performs well. The reason is that the airline domain uses few homonyms and that most interfaces follow similar structural patterns. This enables the algorithm to fully exploit the potential of all its steps.

Auto The auto domain contains small interfaces and uses a comparably small vocabulary with not many homonyms. Therefore, the algorithm performs exceptionally well in the auto domain.

Book The book domain contains interfaces that use a comparably small normalized vocabulary. The algorithm performs well in the book domain.

Job The performance in the job domain is comparably worse than in other domains investigated. Reasons are different design patterns and the widespread homonymous usage of two central concepts. These two concepts are `job type` and `location`. `Job type`, on the one hand, refers to a selection list where a user can decide whether she is interested in a full time or a part time employment. On the other hand, a field labeled `job type` enlists industry areas of the offered jobs. Partially overlapping value sets prevent value-based distinction between the possible two semantics. The second main homonym in this application domain is `location`. Most of the investigated sites use it synonymously to `state` referring to the US-state

where the job is offered. In other interfaces, `location` denotes a more general concept and is modeled as an internal node in the schema tree. Those nodes contain, among others, the state information among the children of `location`. Also, the job domain contains interfaces with fields that share concepts. For example, there are interfaces that offer three fields to enter a city. By design, we are not able to map those interfaces correctly because we assume semantically distinct fields in an interface.

Real Estate The interfaces of the real estate domain are very heterogeneous. First, they differ in size. The smallest interface investigated contains only three fields whereas the largest contains about 30 fields.

Moreover, real estate interfaces fall into three categories. The first category offers only renting arrangements, the second category only properties to buy and the third category contains both options. This makes matching difficult.

Also, value ranges are central on real estate interfaces, such as `size range`, `price range` or `year range`. The range information is encoded in multiple ways. Some interfaces provide a single text field, others offer separate fields for a minimum and a maximum and others use fields and qualifiers. Ranges are mostly grouped. The correct mapping of the internal nodes that represent such a group and that of the grouped fields in some cases fails because both, internal nodes and fields, use similar labels and names. Also, we observed a high number of homonyms among the labels and names of these fields.

Carrental The matching of the carrental domain performs the worst. Event though, the interfaces follow mainly the same pattern. There are many mandatory fields, thus the heterogeneity in structure is limited. But this domain contains three classes of problems that strongly impact the matching results.

First, there are frequently used homonymous labels such as `pickup` and `dropoff`. In some interfaces, these terms refer to a date, in others to a group of date and a time and a third kind of interfaces denotes the rental locations with them. A field with the meaning of one of them occurs on most of the interfaces. Homonyms like these examples reduce the precision because the matching algorithm constructs wrong mappings from homonymously labeled nodes.

Second, the labels and names of the domain contain many synonyms. Undiscovered synonyms reduce the recall because mappings are missed by the matching algorithm. A frequent example of a synonym pair in the carrental domain is `start` and `pickup`. This is not a general purpose synonym, therefore we cannot discover it by using synonym databases such as WordNet.

Third, there are many nodes in the schema trees of this domain that do not contain any label. The matching algorithm has to rely on other parameters such as values or children to infer mappings.

Hotel Interfaces in the hotel domain often follow common design patterns. Therefore, we achieve a good performance. The main problem for the matching is the

homonymously used concept `guests`. Some interfaces use it to denote the number of persons intending to stay in a hotel. Others distinguish finer, for example, they introduce `adults` and `children` such that `guests` is a hypernym of both and therefore, it is attached to the corresponding internal node. A third category of interfaces introduces another field `rooms` to denote the number of rooms to be booked. These interfaces group the three fields named `adults`, `children` and `rooms` and name the group `guests`.

Summarizing we note, that the results of our matching algorithm strongly depend on application domains. Homonymously used terms are the main problem for the matching although the investigation of the structural aspects of query interfaces while matching reduces their impact. Probably, user interaction strongly would improve the results. Domains that use a stricter vocabulary receive much better results than those that contain many homonyms. We noted, that even a small number of frequently used homonyms has a strong impact to the results and may drop the f-measure significantly.

4.6 Discussion

In this section we discuss limits of our approach and give suggestions how to circumvent these limitations. We identify several problem classes that may lead to improper matching results:

Homonym Labels or Names The algorithm does not resolve homonyms directly, therefore it may miss mappings between homonymously labeled elements. Examples for homonyms in the dataset are the terms `from` and `to`. They may denote an airport, or they may be part of section that describes some range parameters such as a price range or year range. Even though, we do not provide a direct method for the resolutions of homonyms, the mapping of the parent nodes of homonymously labeled nodes may help for their disambiguation. In the example of a range information, a correct mapping of the parent node may lead to the correct resolution of the range elements.

Granularity Shifts In some domains, a single concept may be used in different granularities. An example is the concept `guests` in the hotel domain (see above).

The algorithm could resolve particular cases of this problem when introducing a specific handling of mappings between internal nodes and fields. We may either suppress these cases or, alternatively, decrease their similarity score due to the structural differences.

Multiple Concepts in one Field Our model of Web query interfaces assumes that a single field in a query interface denotes a particular semantic concept. Not all interfaces follow this intuition. There are cases where the user may enter different information into a single field. For example, some interfaces of the book domain contain a field where a user either can fill in an authors name or a keyword. We

cannot correctly identify this kind of mappings, because our model does not allow complex mappings or alternative labels.

We may establish a preprocessing of labels that looks for signal terms such as `or`. In case such a signaling term has been identified, the field should be internally split along its meanings.

Limits of Field Order Similarity The order similarity is of limited strength. Especially, when the interfaces to compare do not share some common structure the predicate may compute the wrong order assumptions. For example, in the hotel domain there are interfaces that offer only a `checkin date` while most query interfaces contain both, `checkin date` and `checkout date`. The relative position of the `checkin date` field in the further kind of interfaces may match that of the `checkout date` field of the latter kind of interfaces. This may lead to wrong mappings between those fields.

The order similarity may be redefined using a more complex model for computing the relative order of an element inside an interface. For example, the relative position score may also include the level of the element in the interface tree and other structural properties.

Fields with partially overlapping Semantics Some interfaces provide fields whose semantics overlap only partially. An example is `year` in the auto domain. Some interfaces provide only a single field for `year`, others give a `year range` encoded into two fields. It is unclear, how to map these fields correctly. We see two possibilities: Either we map the single `year` field to the `start year` field in the range representation or, we map the single `year` field to the internal node representing the `year range`. We decided for the later option, but this decision excludes some cases.

Multiple Fields represent the same Concept Some interfaces provide multiple fields to fill in a particular information. For example, in the job domain there are interfaces that allow the user to either select a city from a list or, alternatively, they provide a free text field to fill in this information. Fields in this case should be mapped 1:m. We do not support that kind of fields, instead we pick one of the fields for the mapping.

A better solution could introduce additional internal nodes that group these fields. Because theses groups carry a different meaning than those we identify currently, this step requires to semantically tag the edges of the schema tree. Here, the suitable edge annotation would be `or`.

Summarizing, we observe that our solution contains particular weaknesses. Most of them may be improved by extending the algorithm as described before. Generally, we recommend to use the automatically generated mappings a suggestions. A human expert should review the computed results to ensure their correctness. This procedure follows the commonsense understanding in information integration as most known integration

techniques allow the user to adjust integration steps [53]. Therefore, our tool VisQI (see Chapter 6) provides a comfortable user interface to manually correct the suggested mappings.

4.7 Related Work

Wang [82] proposes an instance-based approach according to the matching classification by Rahm and Bernstein [73]. It matches query interfaces and the results of queries. The system represents the schema of the (hidden) Deep Web source by two interface schemas, the query interface schema and the result schema. It uses instance based matching techniques to match those two schemas. The algorithm poses predefined queries (containing instances) through multiple interfaces and matches their result appearance (position in the result interface) with the appearance on the query interface. The algorithm counts co-occurrences of particular instances of the interface schema and the result schema. It uses a mutual information measure to determine the correct matches. This approach also can be applied to multiple sources. In contrast to our approach [82] uses information from result pages. Their matching quality relies on the quality of the extraction of these result pages which is a quite difficult problem. Our approach does not need result pages at all. The experiments reported in [82] are based on an interface set out of the two domains auto and book. The authors report a precision of 0.94 and a recall of 0.74 for the book domain. In that domain our algorithm performs almost similar. Whereas our precision (0.88) is slightly worse, our recall is slightly better (0.88). For the auto domain our algorithm slightly outperforms the approach in [82]. Whereas Wang proposes a precision of 0.95 and a recall of 0.77 we obtained a precision of 0.96 and a recall of 0.81. Please note, that the interfaces of auto and book domains are generally simple, most of them do not contain any hierarchical structure because [82] is not able to handle more complex interfaces.

He's approach, [34], is based on co-occurrences of schema attributes. It formulates two observations: First, correlated attributes in an interface are likely to be grouped together (positive correlation). Second, it is unlikely that attributes and synonym attributes occur together in a single interface (negative correlation). These two correlation types enable the algorithm to find mappings. First, groups are discovered, second, negative correlations are identified. Third, matching groups from the set of negative correlations are selected. Those groups represent mappings. In contrast to [34], we use correlation assumptions in a different way. We utilize a kind of correlation assumption between terms only when constructing the dictionary (compare Section 4.2). We never correlate other aspects of fields such as their values. The comparison of the accuracy of [34] to ours is difficult because the authors of this work only consider frequent attributes and exclude outliers. Therefore, the authors report very high accuracy values that mostly outperform our results. For example, in the hotel domain they obtain a precision of 0.86 and the recall of 0.87 when having previously pruned those attributes that occur on less than 10% of the interfaces.

The system [33] does not use pair-wise mappings between attributes, but takes a

holistic approach to match all input schemas at once by finding an underlying generative schema model. It exploits two main observations. First, with the growing of the Web, ample sources provide structured information about the domains. Second, while the sources proliferate, their aggregate schema vocabulary remains small. Our approach follows the idea that many interfaces of a domain follow a common structure to a certain extend. We use this idea when comparing internal nodes and there ancestor structures for the purpose of finding mappings. In contrast to [33], we do not generalize this idea to the extend of full domains. Similarly to [82] this work uses only domains whose interfaces do not contain complex hierarchical structures. The only domain that is shared with our dataset is the auto domain. The reported precision of that domain is 0.94 and recall is 0.84, respectively. Our approach outperforms [33] for both measures.

In terms of the matching classification given in Section 2.1.2 Wu's system [91] is a hybrid system. It computes a similarity value composed from a label similarity, a value similarity and a name similarity. This approach also exploits transitivity among the mappings. It defines additional to 1:1 mappings also 1:m mappings between field nodes. This is necessary to correctly map interfaces of different granularity. For example, consider two interfaces of the airline domain. The first interface may contain a field `passengers` denoting the number of passengers traveling, whereas the second interface provides two fields `adults` and `children` instead. In Wu's approach (and others), the correct mapping between these fields is 1:m, because `passengers` maps `adults` as well as `children`. We avoid 1:m mappings between field nodes by considering also internal nodes as targets of a mapping. This allows the reduction of most cases of 1:m mappings on field level to 1:1 mappings on the level of internal nodes. In the above example, we would search for an 1:1 mapping between the group of the fields {`adults, children`} and the field `passengers`.

The experiments of [91] contain interfaces out of the five domains airline, auto, book, job and real estate. The reported f-measures range from 0.88 in the real estate domain to 0.93 in the book domain and average 0.90. All numbers reported in [91] exceed our results by up to 100%. Unfortunately, the description of the experiments in [91] leaves out the details on how the authors picked the interfaces for their experiments and computed the numbers. We believe that the measuring methods are different to ours and a direct comparison of the results would lead to wrong conclusions. Instead, we compare our greedy solution that adopts central ideas of [91] to our setting with our final approach using the Hungarian Method. The comparison shows that our final approach performs about 5% better than the greedy solution (compare to Section 4.5.3).

Finally, the most recent work in the field [40, 39] considers only field-mappings. They do not investigate complex attributes. The approach is restricted to those field mappings that match 1:1. It uses a subset of our dataset, but the reported accuracy values are not comparable with that of our solution due to the different setting.

There is also a qualitative difference between our understanding of mappings and that of most of the related work. We emphasize this difference on an example. Consider the following situation: In the job domain, interfaces contain information about the job location. Assume two interfaces A, B of this domain. Interface A contains three fields A_{state}, $A_{country}$ and A_{city} denoting the country, state and city of the desired job location

and an internal node A_{loc}, labeled `location`, that is parent node of these three fields. Interface B contains only a single field B_{loc} that is named and labeled `location`.

All given previous approaches define mappings between fields only. The gold standard thus would contain the mappings (A_{state}, B_{loc}), $(A_{country}, B_{loc})$, (A_{city}, B_{loc}). The gold standard of our approach for this example contains only a single mapping (A_{loc}, B_{loc}), but together with the ancestor-relationships it provides the same information.

It is interesting to investigate the quality of this two different approaches with respect to further usage scenarios. One common usage scenario of mappings is the merging of interfaces [25]. As long as only 1:1 mappings occur, both approaches behave similar. In the case of complex mappings as in the example above the merging will expand the interface on the m-side of the mapping to a set of fields. This reduces all 1:m mappings to 1:1. In the example above, interface B is expanded such that B_{loc} becomes an internal node with new children B_{state}, $B_{country}$ and B_{city}. In our approach, the information of the expanding of B is already modeled by the tree structure of A and the mapping (A_{loc}, B_{loc}). Thus, for merging purposes our approach provides a better quality of information than field-based approaches.

Another application scenario is the construction of a unified query interface from a bunch of integrated interfaces. Here, a central problem is the labeling of the fields and internal nodes of the unified interface [26]. For this application scenario, our approach performs qualitatively better, because during matching we utilize internal nodes of the query interfaces and we provide also the labels for those internal nodes. Field based approaches miss this information.

There is a principal disadvantage of our mapping approach compared to the related work that considers 1:m mappings. Our approach would miss to find the correct mappings when a particular node of a query interface A is not reflected by a field or internal node of another query interface B, but a group of fields in B maps. Previous approaches are able to find these mappings. We figured out, this case is very rare.

Our approach qualitatively outperforms the field-based mapping approach when interfaces contain field dependencies. An example for field dependencies are qualifier fields. Qualifier fields allow the user to specify or refine the intended meaning of other fields. In the real estate domain some interfaces query a size range information for properties. For example, an interface A may provide a field A_{SF} `Price range` and an adjacent selection box A_{SC} that allows to select one of the three possible values `less, above, equal to` to qualify the field. Usually such related information is grouped. Let A_S be the internal node that represents the group. If, for example, the user selects `less`, the meaning of A_S is `maximum size`. Assume a second interface B defining the size by a range representation. B contains two fields B_{Smin} and B_{Smax} for the minimum and maximum size. Moreover, let B_S be the internal node that represents the group of B_{Smin} and B_{Smax} A field based approach maps both B_{Smin} and B_{Smax} to A_{SF}. Applications that reuse the mapping information, such as merging tools, will likely produce wrong results. In our mapping approach, we define only one mapping between A_S and B_S. The particular meaning of A_S is left open at matching time. This prevents wrong implications from the mappings.

5 Query Interface Domain Classification

We understand by *domain classification* the automated process of annotating Deep Web interfaces with their application domain. Domain classification is a mayor issue for the integration of Deep Web sources. Especially when integrating on the fly, as proposed in [38], the integration system demands a strong classifier that filters out interfaces that do not belong to the domain of interest. As emphasized in Section 2.3 it is important also when performing focused crawling [7].

Domain classification is challenging because of the homonymous usage of terms. On the one hand, interfaces of different domains use similar terms for different purposes. On the other hand, there may be such homonymy even inside a particular domain. Therefore, a straightforward approach that, for example, considers only the terms on the interfaces for domain classification is not promising.

We instead propose methods that fully exploit the results of the previously investigated interface extraction and matching steps. Therefore, our domain classification approach for Deep Web sources combines structural as well as terminological information of the interfaces.

We discuss two different approaches for domain classification. In Section 5.1, we introduce *domain patterns* to define a pattern-based domain classifier. Therein, we cast the domain classification problem into a pattern matching problem. In contrast, in Section 5.2, we derive an *interface similarity measure* from the element similarities introduced in Chapter 4 and define an alternative classifier based on this measure. We transform the classification problem into a nearest neighbor search problem. We evaluate both approaches in an experimental study in Section 5.3. In Section 5.4 we compare both approaches and discuss their advantages and disadvantages. Finally, the related work is presented in Section 5.5.

5.1 Pattern-based Classification

A domain pattern describes central aspects of a particular domain by reflecting main concepts and the general structure of a relevant portion of the interfaces in the domain. We can derive a useful domain pattern from a sample set of interfaces if (1) the number of *relevant* concepts in a specific application domain is rather small and (2) the elements on the query interfaces can be clustered to their concepts by exploiting similarities among their used terms, structures and layouts.

We separate the pattern-based classification process into the three steps domain pattern construction, domain pattern matching and domain assignment.

5.1.1 Construction of Domain Patterns

Prior to the definition of domain patterns we introduce *concept sets*.

Definition 5.1 (Concept Set) *Let C be a set of Web query interface elements, each from a different query interface. C is called to be a* concept set *if all elements of C belong to the same semantic concept.*

For an example of concept sets, consider the two interfaces in Figure 4.2 on Page 67. Both interfaces, A and B, contain fields for departure day information (fields $depDay_A$ and $depDay_B$) as well as return day information (fields $retDay_A$ and $retDay_B$). Because `departure day` and `return day` represent two different concepts, we would like to construct two concept sets from the four elements, i.e. $c_{depDay} = \{depDay_A, depDay_B\}$ and $c_{retDay} = \{retDay_A, retDay_B\}$.

We obtain the concept sets by first computing mappings between query interface elements as described in Chapter 4. Two elements s and t belong to the same concept set, if there is a path of mappings from s to t or vice versa.

Based on concept sets we now describe a domain pattern as an ordered subset of those concept sets whose cardinality exceeds a particular threshold. Therein, the order among the concept sets represents the structural properties. More formally we define:

Definition 5.2 (Domain Pattern) *Let CE be the set of all concept sets C of a particular domain, let further $\prec$ be an order relation and let τ be a threshold. Then, a domain pattern P is defined as:*

$$P = \{C | C \in CE \wedge |C| > \tau \wedge \forall C_i, C_j \in P : i < j \Rightarrow C_i \prec C_j\}$$

.

We compute the order of the concept sets in a domain pattern by averaging the normalized postorder ids of the contained interface elements. The postorder ids are obtained directly from the schema trees.

Algorithm 5.1 describes the construction of a domain pattern from a set of interfaces in the particular domain. The input of the algorithm are a set of elements E from a particular domain, a set of element mappings M and a threshold τ. The output is a domain pattern P containing of a number of concept sets C. The central part of the algorithm is the concept set computation. Therefore, the algorithm iterates over all elements $e \in E$. Initially, each single element forms one concept set. Next, the algorithm adds elements to C that fulfill the following properties: (1) they are part of a mapping $m \in M$ and (2) their mapping partners are already in C. This step is repeated as long as C changes. If the size of the current concept set C exceeds a threshold τ it is added to the domain pattern P.

Algorithm 5.1 PatternConstructor (E, M, τ)

Input: a set E of interface elements, a set M of element mappings, a threshold τ
Output: a domain pattern P

```
P := ∅;
for all e ∈ E do
  C := {e}
  while C changes do
    C := C ∪ {e'|∃(e, e') ∈ M ∨ ∃(e', e) ∈ M}
  end while
  if |C| > τ then
    P := P ∪ {C};
  end if
end for
return P;
```

5.1.2 Matching Domain Patterns

In the context of domain patterns we understand by *matching* the process that identifies whether a particular pattern reflects the main properties of a given query interfaces. The matching generally is computed bottom up, starting from the query interface elements. Therefore, we first define the concept similarity of a query interface element e and a particular concept set C. This measure can be used to distinguish whether e is an instance of the concept described by C.

Definition 5.3 (Concept Similarity) *Let e be an interface element and C be a concept set. The element concept similarity $ConceptSim$ is defined: $ConceptSim(e, C) = \sum_{c_i \in C} LocalSim(e, c_i)$*

The computation of the concept similarity utilizes the local similarity *LocSim* as introduced in Chapter 4 (see Definition 4.11 on Page 76).

Now, we can assign an element e to its semantical concept. The underlying intuition is that the elements of a single semantic concept follow some common structure. Therefore, the concept similarity should be maximal if e is of the concept described by the elements in C.

Definition 5.4 (Element Concept) *Let e be an interface element and CE be a set of concept sets. The element concept is that concept set $C \in CE$ that has the best normalized aggregated element similarity with e:*

$$concept(e) = \{C | C \in CE \wedge ConceptSim(e, C) = \max_{C \in CE}\}$$

Next, we define the matching of a query interface and a domain pattern. Even though, interfaces of a domain overlap semantically, they are highly heterogeneous. Therefore, we do not claim a complete matching such that each element in a particular query interface has a mapping concept in the corresponding domain pattern. Instead, we characterize a matching of an interface and a domain pattern by two criteria: (1) at least two concept sets of the pattern should be reflected by the interface, (2) the order of the matching elements should meet the order of the matching concept sets.

Following this intuition, we define the matching of a query interface to a pattern as follows:

Definition 5.5 (Domain Pattern Matching) *Let $Q = \{e_1, ..., e_m\}$ be a query interface with elements e_i and let $P = \{C_1, ..., C_m\}$ be a domain pattern. Then, the matching PM between Q and P is defined:*

$$PM = \{(e, C) | e \in Q, C \in P : concept(e) = C \wedge \\ \forall (e_1, C_i) \in PM \exists (e_2, C_j) \in PM : e_1 \neq e_2 \wedge C_i \neq C_j \wedge (e_1 \prec e_2 = C_i \prec C_j)\}$$

5.1.3 Domain Assignment

Having defined patterns and their matching, we now describe the pattern-based domain classification.

We define a domain pattern match ratio that is a measure for the quality of a particular matching. The pattern match ratio is used to quantitatively compare multiple matchings. It aggregates the concept similarities of all elements of the matched particular query interface with the concept sets of the domain pattern.

Definition 5.6 (Pattern Match Ratio) *Let P a domain pattern and Q be a query interface. Let further PM be the matching of P and Q. The domain pattern match ratio r is defined:*

$$r(PM) = \begin{cases} 0 & PM = \emptyset \\ \frac{1}{|PM|} \sum\limits_{(e,C) \in PM} (ConceptSim(e, C)) & else \end{cases}$$

In our understanding, every domain has a corresponding domain pattern, therefore we transform the domain assignment problem into a pattern selection problem. We represent the set of domains available by a set of domain patterns D.

Let Q be an unknown query interface and $D = \{P_1, ..., P_n\}$ a set of domain patterns. Initially, we compute for each relation (P, Q) the corresponding domain pattern matching together with its matching ratio.

We distinguish two scenarios for the domain assignment to Q. In the first scenario, we assume that Q belongs to one of the previously learned domains. In this case, D contains a pattern that represents the domain of Q. We select the domain pattern $P_{best} \in D$ that matches Q with the highest matching ratio and assign the domain of P_{best} to Q. In the rare case that a particular interface matches two or more different patterns with equal ratio we randomly pick one.

In the second scenario, we assume that Q may belong to a domain that has not been learned previously. In this case, we introduce an additional threshold to define a minimum matching ratio. We use this threshold to sort out interfaces that do not belong to any of the present domains.

5.2 Neighbor-based Classification

We also developed a second, *neighbor-based* domain classifier to compare the results to that of the pattern-based classification. This classifier does not utilize computed patterns. Instead, it directly uses the similarity of interfaces.

Neighbor-based classification assumes that those interfaces that contain similar elements and have a similar structure most likely belong to the same domain. Therefore, the classification problem is transformed into a similarity computation problem on interface level. More in detail, neighbor-based classification estimates the application domain of an interface in a two step process. First, it calculates pairwise *interface similarities* between the interface Q to be classified and all interfaces of the learning set. Second, it selects the interface N that is the closest *neighbor interface* of Q according to the computed similarity. Third, the algorithm assigns the domain of N to Q. In the case that there are more than one interfaces N obeying the same similarity, we pick one of them randomly and assign its domain to Q. In our experiments we never witnessed such a case, therefore we decided not to investigate this rare case more in detail.

In the following, we first define similarity measures for interface comparison. We then develop the classification algorithm and finally describe its evaluation.

5.2.1 Computation of the Interface Similarity

We denote by interface similarity a measure to approximate the semantical closeness of two interfaces. We build up the interface similarity on mappings between interface elements (compare Chapter 4). We introduce different viewpoints to describe the similarity of two interfaces. We define measures that implement these viewpoints.

Viewpoints

The intuition of a good match between two interfaces is not clear in general. Therefore, we introduce three different viewpoints. We call the first viewpoint *containment*. Con-

sidering only this viewpoint, two interfaces match perfect, when one of the interfaces is completely contained in the other one. The second viewpoint, *distance*, emphasizes the difference of two interfaces. Considering this aspect separately, two interfaces match best, when the number of non-matching elements is minimal. The third viewpoint, *quality*, is based on the similarity scores of the interfaces element mappings. Considering only this aspect, those interfaces match best, whose aggregated element matching scores are maximal.

As an example, consider two airline interfaces A and B that share all the mandatory fields such as `departure airport` and `arrival airport`. Assume further, that one of them, B, offers an additional `airport` field to allow the booking of a multi city trip. When considering only the containment viewpoint, the matching of A and B is perfect because A is completely contained in B. In contrast, we have to discard the matching when considering the distance viewpoint. The reason is the missing optional field in interface A. Finally, the quality viewpoint investigates aggregated accuracy and therefore may lead to both conclusions depending on the element similarity scores.

Interface Similarity Measures

We define measures to quantify the above introduced viewpoints. We distinguish between *simple methods* and *combined methods*. Simple methods consider only one of the viewpoints whereas combined methods combine at least two viewpoints. We define three simple measures: $ContSim$ for the containment viewpoint, $DistSim$ for the distance viewpoint and $QSim$ for the quality viewpoint.

Definition 5.7 (Simple Interface Similarities) *Let S and T be two interfaces and let M be a set of mappings among the elements of S and T. We define the* containment similarity *$ContSim$, the* distance similarity *$DistSim$ and the* quality similarity *$QSim$ as follows:*

$$ContSim(S,T) = \frac{|M|}{min(|S|,|T|)}$$

$$DistSim(S,T) = \frac{2 \cdot |M|}{|S| + |T|}$$

$$QSim(S,T) = \frac{\sum_{m \in M} NodeSim(m)}{|M|}$$

We investigated different combinations of these three measures (data not shown). We found those combinations most promising that utilize one of the first two simple measures and the quality similarity. We achieve the following two combined measures:

Definition 5.8 (Combined Interface Similarities) *Let S and T be two interfaces and let M be a set of mappings among the elements of S and T. The* containment

quality similarity *ContQSim and* distance quality similarity *DistQSim is defined as follows:*

$$ContQSim(S,T) = \frac{\sum_{m \in M} NodeSim(m)}{min(|S|,|T|)}$$

$$DistQSim(S,T) = \frac{2 \cdot \sum_{m \in M} NodeSim(m)}{|S| + |T|}$$

5.2.2 Classification Algorithm

Algorithm 5.2 NeighborClassifier(L, Q, τ)

Input: a Learning Set of Interfaces L, an interface to classify Q, a threshold τ
Output: a domain class for Q

```
L_neighbor := null;
for all L_i ∈ L do
  if Sim(L_i, Q) > Sim(L_neighbor, Q) then
    L_neighbor := L_i
  end if
end for
if Sim(L_neighbor, Q) > τ then
  return domain(L_neighbor);
else
  return null;
end if
```

Algorithm 5.2 implements the neighbor-based classifier. The algorithm takes as input the preclassified learning set L of interfaces from several domains, an interface Q to be classified and a threshold τ. It iterates over all interfaces $L_i \in L$. The algorithm computes for each pair (L_i, Q) a similarity score using the function *sim*. This function is a proxy for one of the above defined similarity measures (compare Section 5.2.1). The nearest interface is stored ($L_{neighbor}$).

Having iterated over all interfaces of the learning set the algorithm returns the domain of the nearest interface $L_{neighbor}$ if the similarity score is above a threshold τ or *null* if not. We use τ to avoid false classifications if the similarity score is too low and a particular interface may not belong to one of the predefined classes.

The selection of the concrete similarity function and the threshold adjustments will be discussed in Section 5.3.

5.3 Evaluation

In this section we evaluate both classification algorithms. We first describe the experimental setting and then perform the same experiment for both algorithms. We finally compare the results of both experiments.

5.3.1 Experimental Setting

The dataset is similar as that described in Chapter 4: It contains 140 interfaces of seven domains. We perform multiple "leave-one-out experiments" to compute an average accuracy value. Therein, we pick one interface, learn the domains using a subset of the remaining interfaces and apply the classification algorithm for the selected interface. The procedure is repeated for all interfaces for cross validation.

We define two scenarios for the classifier evaluation, *full domain knowledge* and *limited domain knowledge*. Both differ in the structure of the learning set.

Full Domain Knowledge The full domain knowledge scenario assumes that the interface to be classified belongs to one of the known domains in the learning set. Therefore, we learn with interfaces of all known domains.

Limited Domain Knowledge The limited domain knowledge scenario assumes that the interface to be classified may belong to another domain than those previously learned. In this scenario, we performed 7 different experiments in the same style as described before. For each of them, we left out all interfaces of one particular domain while learning. Therefore, for each classification, there is always a chance that the interface to be classified belongs to the domain not learned in the current experiment.

5.3.2 Evaluation of Pattern Classifier

Adjustments

We use two thresholds for the application of the pattern classifier. The first threshold we name element threshold τ. This threshold distinguishes whether a particular query interface element matches a concept set or not (compare Definition 5.1). The second threshold, the pattern threshold T, is used by the classification algorithm to decide whether a particular interface matches a domain pattern or not.

In our experiment we adjusted both thresholds globally using a small separated portion of the learning set. For this portion we estimated the best values as $T = 0.1$ and $\tau = 0.5$. We also investigated a domain dependent adjustment of both thresholds. Our experiments showed, that the performance did not increase (data not shown).

Experimental Results

Figure 5.1 shows the results of the pattern classifier experiments. The leftmost bar represents the full domain knowledge experiment. The classifier correctly classifies 78%

of all interfaces. The other seven bars show the performance of the classifier when the learning set was incomplete. We see a drop in performance by about 10% compared to the full domain knowledge scenario. This is caused either by missclassifed interfaces or by interfaces that belong to one of the known domains, but they did not meet the pattern threshold T and therefore did not get a domain assigned.

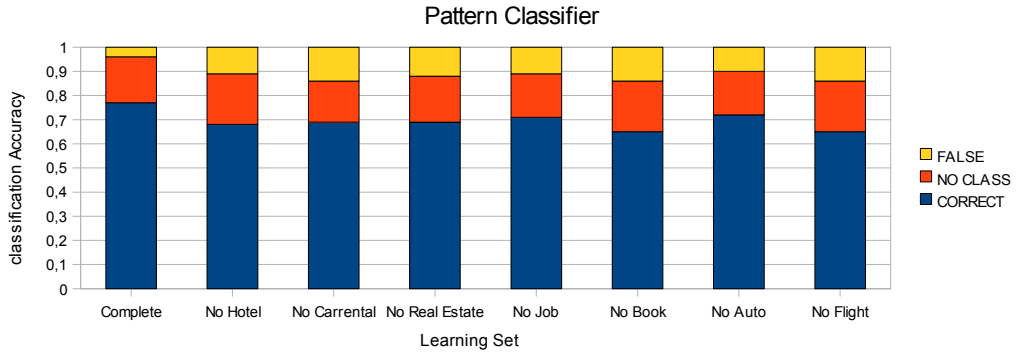


Figure 5.1: Evaluation of Pattern Classifier.

5.3.3 Evaluation of Neighbor Classifier

We also evaluated the neighbor classifier in the same way as described above for the pattern classifier.

Selection of Similarity Measure

We first selected the similarity measure for the nearest neighbor classifier. We separated a subset from the overall dataset and classified several interfaces against this subset using different methods for the computation of similarities. We computed the accuracy of all methods for different thresholds. Figure 5.2 shows the results of the experiment. We can see that two of those similarities that incorporate the containment viewpoint ($ContSim$ and $ContQSim$) outperform the others. Therefore, we selected the containment quality similarity $ContQSim$ for our overall experiment which we describe next.

Experimental Results

Figure 5.3 shows the performance of the neighbor classifier. We can see, that the classifier performs quite well in the full knowledge scenario (about 90% classified correctly). Again, the performance drops by about 10% when the knowledge in the learning set is incomplete.

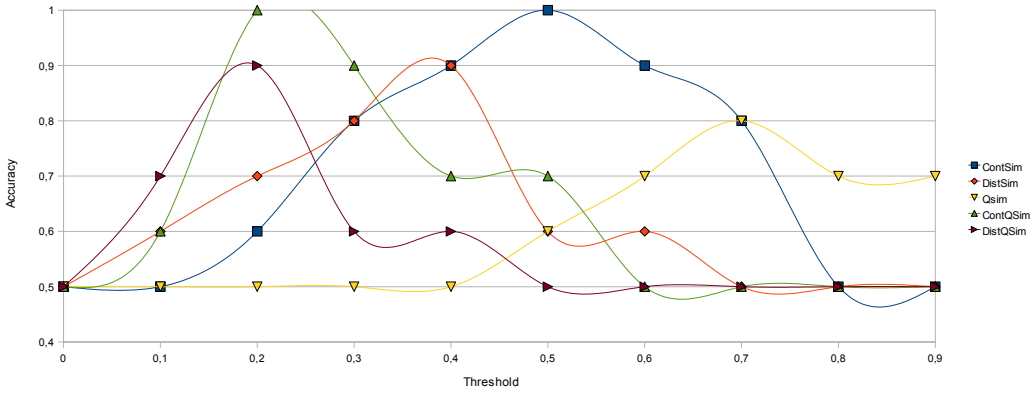


Figure 5.2: Adjustments for nearest neighbor classification.

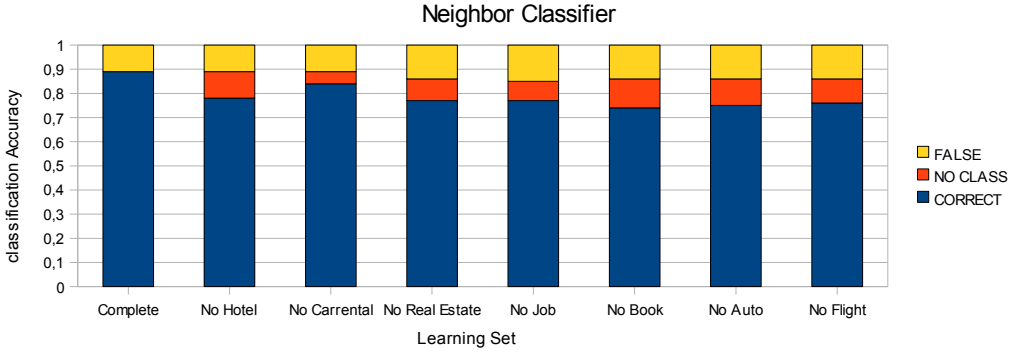


Figure 5.3: Evaluation of Neighbor Classifier.

5.4 Discussion

In this section we relate the results of both classifiers, discuss advantages and disadvantages of both approaches and suggest improvements.

When considering the aggregated results of all domains, the neighbor classifier performs better in all scenarios. This is especially true in those scenarios where the domain knowledge is incomplete. We identify several reasons for the worse performance of the pattern classifier. One reason is, that the classification constraints lack generality. We observed that there are interfaces that belong to a particular domain but they are missed because they do not follow defined domain constraints. Also, the size of the learning set is probably too small to achieve relevant patterns. moreover, the algorithm decides the relevance of an element concept set only by its number of elements. Here, a clustering on a finer granularity, for example by considering also the element similarities, may improve the algorithm.

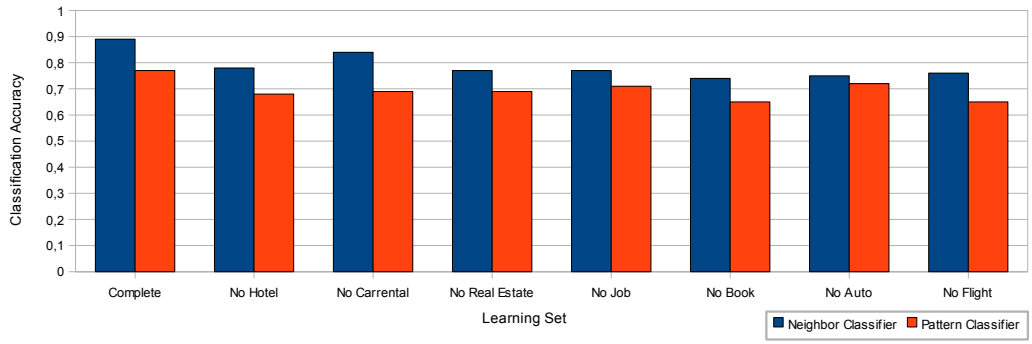


Figure 5.4: Comparison of both Classifiers

We constructed a confusion matrix of both classifiers (Table 5.1) that gives a more detailed view to the false classifications. The matrix relates the true and predicted domains for the interfaces in our dataset in the full knowledge scenario. Each table cell contains the absolute number of interfaces of the domain in the current row that have been misclassified to the domain of the current column where each domain has 20 interfaces in total. The first number in each cell refers to the results of the pattern classifier, the second number to the neighbor classifier.

We observe that the neighbor classifier does not outperform the pattern classifier in all domains. In contrast, in the real estate domain the accuracy of the pattern classifier is higher than that of the neighbor classifier. The reason is that this domain has the highest heterogeneity among its interfaces. Especially the number of fields per interface differs largely in this domain. It is difficult to identify representative interfaces for such a domain. The neighbor classifier may pick a non representative interface and therefore, it may predict an wrong application domain. In contrast to the neighbor classification

the pattern-based approach forces only partial overlappings of the interfaces in a domain and therefore fits better to heterogeneous domains.

We conclude that a combined method that uses both classifiers may be useful. The classification algorithm may select the right classifier based on some statistical information (e.g. standard deviation of fields per interface) in an additional preprocessing phase. In this setting, the pattern classifier should be preferred to the neighbor classifier especially when the interfaces of a domain tend to be heterogeneous.

Beside of this, the classifiers may be improved by incorporating additional information. A potential improvement of the pattern classifier is the integration of interface layout information to describe the patterns. The neighbor classifier may be improved by considering the top-k nearest interfaces instead of only the nearest one.

Domain true/predict.	Airline	Auto	Book	Job	Real Estate	Car-rental	Hotel	No Class	Sum
Airline	-	0/0	0/0	0/0	0/0	1/0	4/2	0/0	5/2
Auto	0/0	-	0/0	2/0	1/0	0/0	1/0	2/0	6/0
Book	0/0	0/0	-	2/0	0/1	0/0	0/0	0/0	2/1
Job	0/0	0/1	0/0	-	3/2	0/0	0/0	2/0	5/3
Real Est.	0/0	1/1	2/1	1/4	-	0/0	0/0	0/0	4/6
Carrental	1/0	0/0	0/0	1/0	0/0	-	2/0	2/0	6/0
Hotel	1/0	0/0	0/0	1/0	0/3	1/0	-	0/0	3/3
Sum	2/0	1/2	2/1	7/4	4/6	2/0	7/2	6/0	31/15

Table 5.1: Confusion matrix of pattern/neighbor classifier.

5.5 Related Work

The three most relevant related works in the field domain classification of Deep Web sources are [62, 5, 35]. Out of them, [35] follows a clustering-based model. This work categorizes Deep Web interfaces along specific criteria given by their schemas. They represent the schema of a Deep Web interface by sets of attributes. Attributes are mainly derived from fields, they correspond to the leaf nodes in our schema tree representation as introduced in Chapter 3. The approach uses a model-based clustering method. Therein, the model is designed as a multinomial distribution and the model differentiation is a objective function of clustering based on statistical hypotheses testing. Similarly to our approach the accuracy of this approach depends on a good extraction. However, this work does not consider any structural information of Web query interfaces and focuses only on of the labels of the attributes. The experimental results of [35] are based on different measures (such as entropy). Therefore, they cannot be compared to our results.

[62] concentrates on e-commerce Web pages with the special focus of comparison shopping systems. The approach models query interfaces as documents of its terms. It uses a vector representation based on the TF/IDF vector-space model [2]. The algorithm extracts features such as price tags or the number of links from the Web pages. A feature

vector is derived for each query interface. This feature vector is used to cluster the query interfaces by their application domain. This approach does not consider values or names, also it does not investigate the structure of the elements on a query interface. The main restriction is the limitation to e-commerce Web sites. The experimental set up for this approach was different from ours. The authors of [35] used Web sites classified by Yahoo and reconstructed their application domains as assigned from Yahoo. The reported F-measures range from 0.86 to 0.94.

Barbosa et. al. [5] follows also a clustering idea. Web query interfaces are modeled as a set of hyperlinked objects characterized by a number of features. More in detail, the approach defines two feature spaces: the contents of the query interface and the contents of the page. It uses a k-means clustering algorithm [59] to find homogeneous clusters of query interfaces. Also, they use Hubs (pages that point to many other pages) to improve the results of their technique. In contrast to our work, the approach uses document clustering techniques and does not infer a query interface schema. It does not disambiguate between labels, names or values but understands a query interface as document and applies the TF/IDF vector-space model [2]. The authors of [5] conduct an experimental study using a subset of the Tel8 dataset, that is comparable with our dataset (for more details about the Tel8 dataset see Section 3.3 on Page 58). The outcome of the experiments showed that the algorithm is able to classify the interfaces of the test set with an overall F-measure of 0.91 to 0.96 depending on algorithm parameters. These results underline that [5] currently is the best available approach for query interface domain classification.

The main limitations of the two document-based approaches [62] and [5] are heterogeneous term vocabularies inside an application domain or homonymously used terms across domains because the vector space model demands distinct term sets (syntactical as well as semantical) to differentiate documents correctly.

Summarizing, our solution currently does not outperform the results of all competing approaches. Nevertheless, there is room for future improvements. For example, a novel pattern model in the pattern-based classification or a better strategy to select representing interfaces in the neighbor-based classification could increase the performance.

6 The Visual Query Interface Integration System (VisQI)

In this chapter, we describe our system VisQI (VISual Query interface Integration system) [43] that provides a graphical user interface for all steps of Web query interface integration as described in the previous chapters. It encompasses our algorithm to extract HTML query interfaces into hierarchical representations, it matches different query interfaces based on mappings of their elements. Finally, the system allows to classify previously unseen Deep Web interfaces to their application domains. VisQI has a sophisticated evaluation component which is accompanied by large gold standard sets for both query interface extraction and matching. An intuitive visualization component eases the work of the user.

The system has been implemented in Java. We use OLE based on .NET technology for the rendering and displaying of Web interfaces. Utilizing these technologies assures that the rendering components can cope with most recent HTML development techniques (e.g., DHTML, XHTML).

Developers may use VisQI as a standalone application. They also may incorporate intermediate results of VisQI (e.g., extracted schema trees, domain-wise mappings) in their own projects. Due to the modular architecture of the system, they may reuse the individual components to build their own systems. Finally, developers may evaluate their own algorithms by VisQI.

Though there has been a considerable number of proposals to Deep Web integration (see, among others, [37, 93]) we are not aware of any comparable solution that is available as running and extensible system that supports all three steps of integration and is equipped with a data set for testing as large as that of VisQI.

6.1 System Components

The system is designed as a framework that consists of loosely-coupled components. The architecture of VisQI is depicted in Figure 6.1. The key functionalities of the system are described in the following.

6.1.1 Rendering and Extraction Component

This component first converts a Web page into a set of *tokens* and, second, it constructs the schema tree from the token set. The component corresponds to the extraction step as described in Chapter 3.

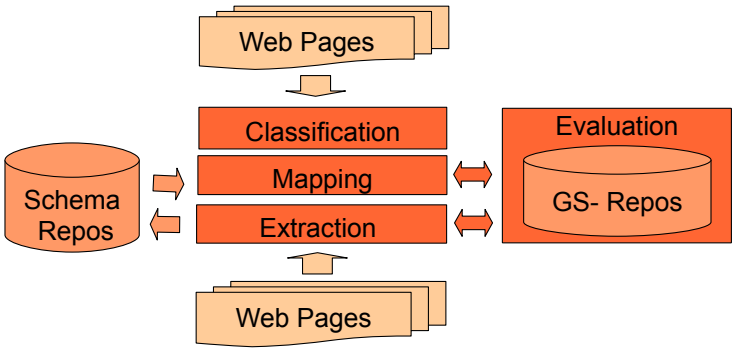


Figure 6.1: Architecture of VisQI

Recall, that a token in our understanding refers to a visible element of the page (e.g., a label or a field) that has an associated list of properties such as a rectangular box describing the visual placement of the element (compare Section 3.2). The rendering component of VisQI uses the various properties of tokens to infer the schema tree (see Definition 2.5 on Page 22). Figure 6.2 visualizes the result of the rendering and extraction step. The extracted schema tree is shown in the middle panel, rendered bounding boxes are shown in the browser view.

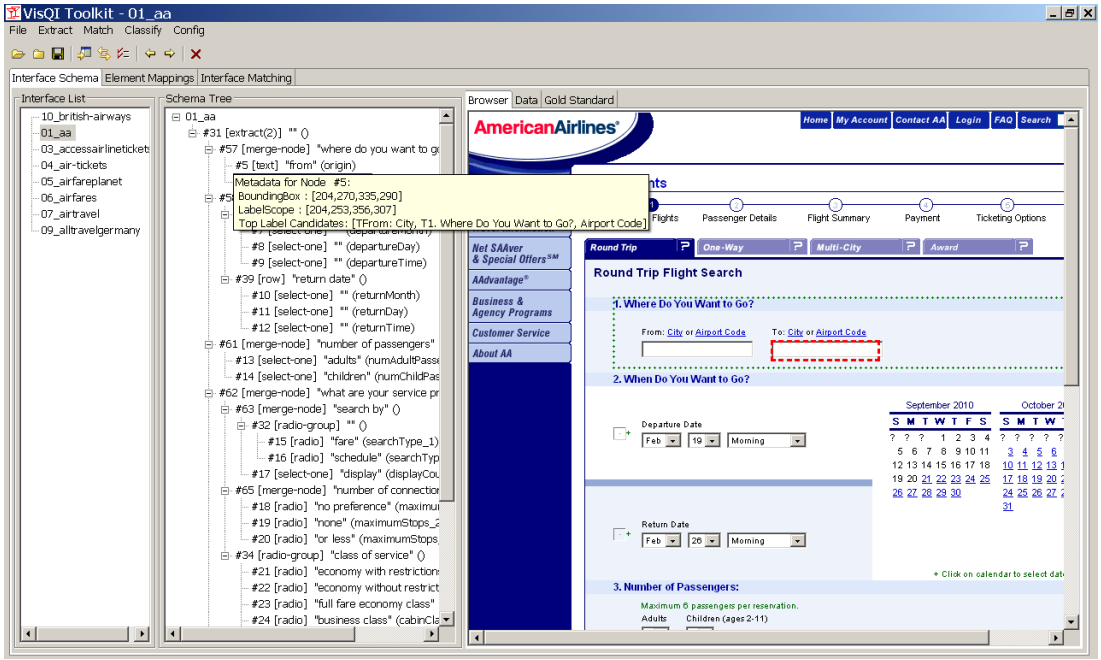


Figure 6.2: Extraction result and visualization of a schema tree and its properties

6.1.2 Matching Component

This component allows to identify semantically equivalent elements on a number of interfaces. The matching component takes as input a list of schema trees and outputs matchings between query interfaces. A matching contains a number of interface element mappings. This component corresponds to the matching algorithm as described in Chapter 4.

Figure 6.3 depicts the mappings between the nodes of two query interfaces. Query interfaces are shown in the hierarchical representation. A line between the nodes of two distinct schema trees represents that the two nodes may be semantically equivalent.

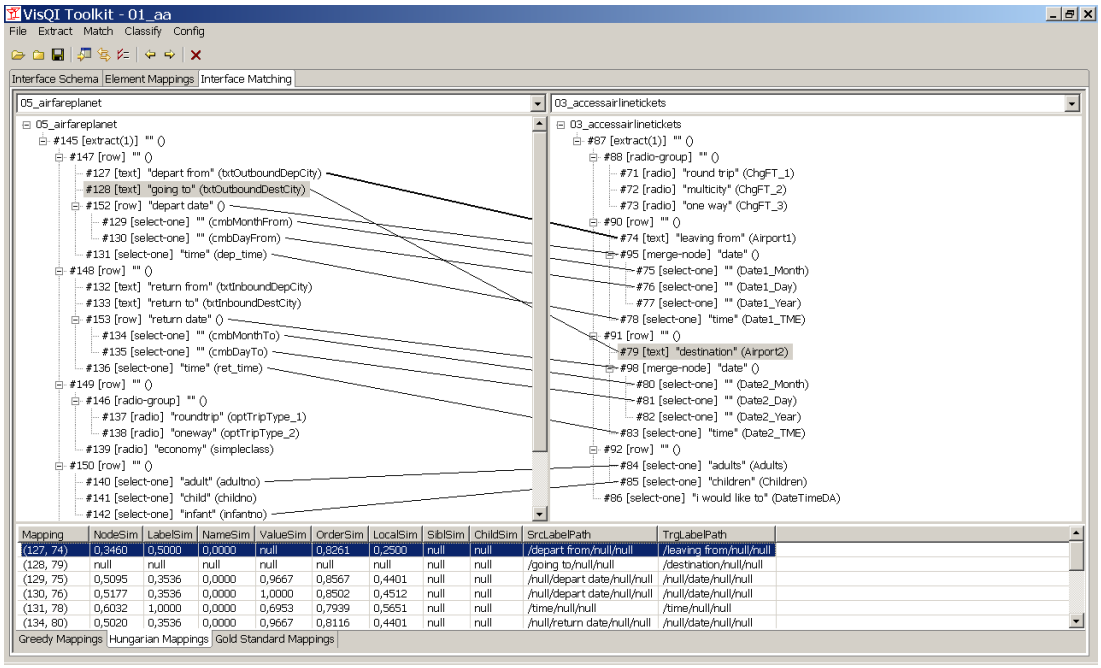


Figure 6.3: Detail visualization of mappings.

6.1.3 Classification Component

This component determines the application domain (e.g., carrental, real estate) of an unknown query interface. It corresponds to the classification step as described in Chapter 5. VisQI implements both classification approaches, neighbor-based classification and pattern-based classification. Figure 6.4 shows the result of a classification process. In the example, the interfaces enlisted in the leftmost panel have been used for learning purposes. Other interfaces have been classified. The classification result is shown in a message box.

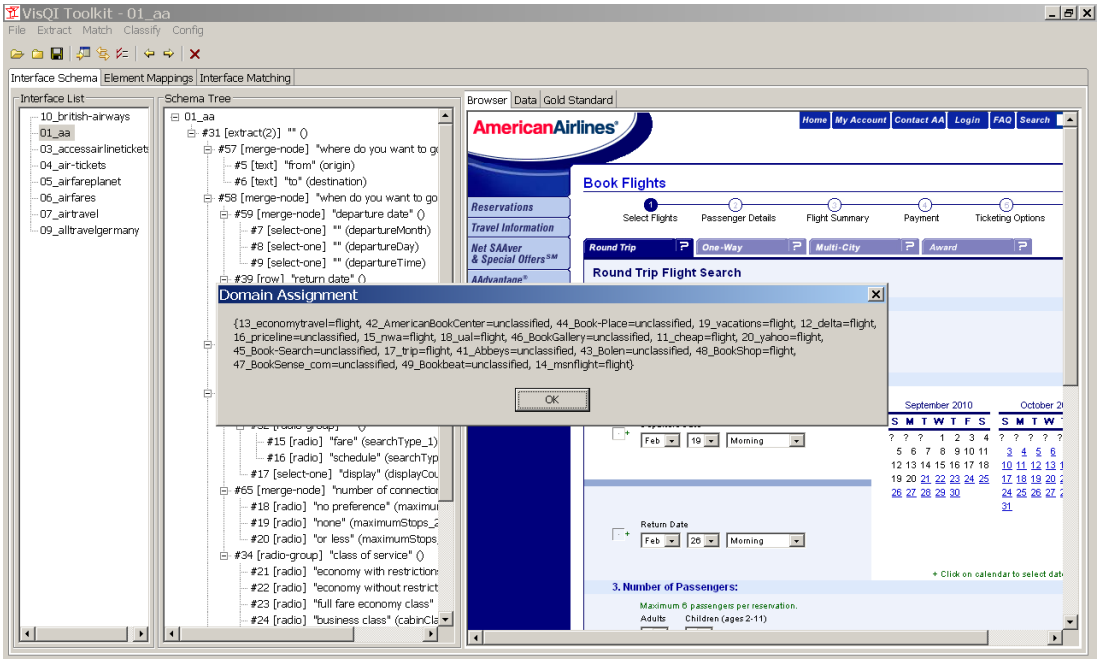


Figure 6.4: Classification of unknown query interfaces to their application domain.

6.1.4 Evaluation Component

This component allows for the assessment of the results of the other components. The evaluation utilizes manually defined gold standards. Currently, the system is equipped with schema trees of more than 500 interfaces in 15 application domains. For about 200 interfaces of the data set, the gold standard provides manually defined mappings among the elements of the interfaces. Please refer for details about the gold standard to Section 3.3 on Page 58 and Section 4.5 on Page 84.

Figure 6.5 shows an evaluation of an extracted schema tree. The extracted tree is shown in the middle panel. The system displays also the corresponding gold standard schema tree in the rightmost panel. It highlights differences in both trees in order to ease a visual inspection of the result.

Figure 6.6 shows an evaluation result after a set of interfaces have been matched. The results of both introduced matching methods are shown. The system computes recall, precision and f-measure for a concrete matching.

6.2 Usage of VisQI

The user interface may display either a single interface (schema view, see Figure 6.2), a set of interfaces and its elements (element mapping view, see Figure 6.6) or the matching between two particular interfaces (interface match view, see Figure 6.3).

In the following, we describe the user interface of VisQI along several typical usage

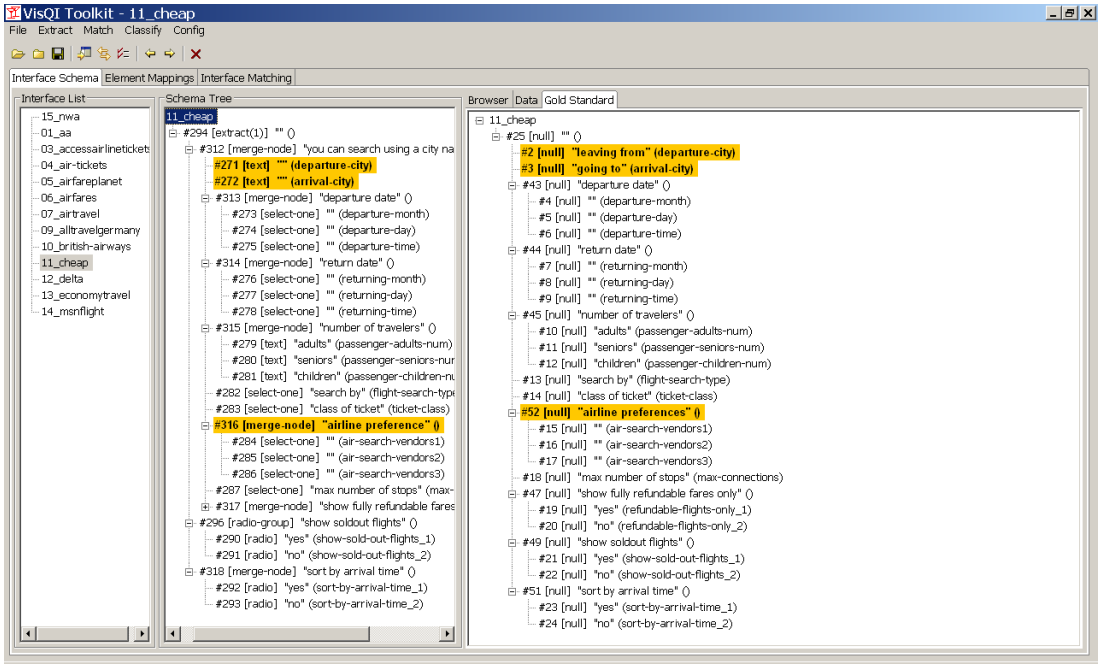


Figure 6.5: Comparison of extraction result side by side to the gold standard.

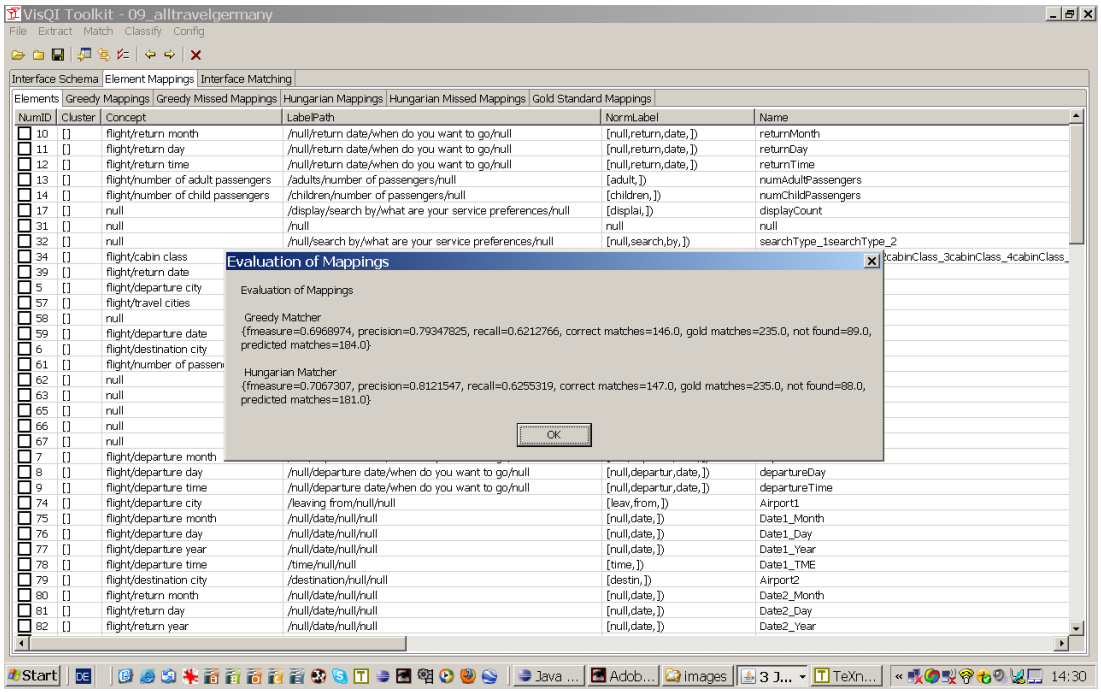


Figure 6.6: Summarized evaluation of matchings and comparison of different matchers.

scenarios.

6.2.1 Rendering and Extracting Web Pages

Although all browsers contain a rendering engine, the visual coordinates and shapes (e.g. semantic scope, bounding box) within query interfaces are quite difficult to access. Many applications, especially Web extraction and integration tools as ours, rely on the graphical rendering information to understand their input pages. Those applications may reuse our rendering component as an easy-to-use module that delivers this information for HTML pages. For details of the underlying algorithm please refer to Section 3.2.1.

The system may be used to extract schema trees interactively or in the background. The result, primarily an XML file containing the tree, can be used as input to other systems/components, such as query interface matching and merging [25]. VisQI can also be used as a tool to debug extraction results. When used in this fashion for a given Web page, the main window of the application has multiple vertical panels. The panel on the left shows the list of currently loaded interfaces. The middle panel shows the hierarchical representation of the currently active query interface as inferred by our algorithm and the right panel contains a tab-folder that holds different views of the current interface such as the *rendered version* of the Web page (see Figure 6.2) enriched with other visual clues. For example, the extracted elements are highlighted, the bounding boxes of the fields and labels are emphasized, etc. During extraction the user can specify either a set of URLs (batch mode) or a single URL.

6.2.2 Domain Classification of Interfaces

VisQI implements a domain classification algorithm which automatically infers the application domain of a query interface. As alluded in the previous section, the intuition of the classification algorithm is that an interface that has "better" matchings with the interfaces in an application domain A than those in an application domain B is likely to belong to A rather than to B. For more details about the domain classification approach please refer to Chapter 5. When used for classification, VisQI may be used in two different scenarios. In the first scenario, VisQI interprets a set of loaded interfaces as learning set for a single application domain. A second set of interfaces is used as test set such that VisQI investigates whether the interfaces of the test set belong to the learned application domain. Alternatively, a user may utilize a set of previously classified interfaces of multiple domains. In this scenario, VisQI is able to predict the application domain out of multiple alternatives.

6.2.3 Matching Query Interfaces and Help at Design of integrated Interfaces

When the system is used for matching, there are two different modes available, the interface matching view or the element mapping view.

In the interface match view, the system displays two particular interfaces in their schema tree representation and visualizes their matching by lines between the corre-

sponding elements. A user can delete incorrect mappings, add missed mappings and edit a mapping. If a matching golden standard is present, the system highlights the incorrect and the missed mappings to a user. This view can also be used to manually define the matching between two query interfaces from scratch. In the lower portion of the view, the system shows the computed similarities for the mappings visualized in a tabular format. For example, for two nodes a and b, their overall similarity score is shown in the column `NodeSim`. If a and b are internal nodes, the column `ChildSim` shows the aggregated similarity score between their child nodes. Figure 6.3 shows the interface match view for two example interfaces.

The alternative element mapping view displays information about interface elements and pairs of elements in a tabular format (see Figure 6.6). It contains two different table types, tables of nodes and tables of mappings. Each row in a table of nodes shows a particular node of the selected schema trees with its properties such as the label or values. In contrast to the table of nodes, a row in a table of mappings represents a pair of candidate mapping nodes from two schema trees. The columns in the table of mappings show multiple similarities for each pair. In contrast to the interface match view the element mapping view allows to compare semantically similar mappings among a bigger number of interfaces.

6.2.4 Developing and Testing of Extraction and Integration Algorithms

VisQI may be used also for visualization purposes without considering the internal matching and extraction components. Visualizing and editing of extracted structures are valuable functions during the development of extraction algorithms. First, developers immediately see the result of their software and can easily identify its failures by looking at the highlighting in the Web page. Second, the user may adjust intermediate results (e.g., the semantic scope of a label, the mapping between the tree of labels and the tree of fields) and then re-run the appropriate components of the algorithm to see the consequences of such changes before having to implement them in the algorithm.

Also, due to the modular architecture, it is fairly easy to replace the extraction algorithm with another one. This feature can be used by every developer to make use of the other components of our software. She simply plugs-in her algorithm into our system, which then displays the extracted interface graphically. By comparing the extracted interface against a gold standard (possibly produced by herself as described above), she can analyze differences as highlighted by our system. This makes testing a lot easier. Alternatively, a developer may partition the gold standard of interfaces into two subsets, one for training and the other for testing.

Although VisQI can perform one interface at a time experiments, this is not practical when the goal is to integrate hundreds of Deep Web sources. The large gold standard set that accompanies the system permits large-scale experiments. A batch mode is implemented to support large experiments, which generates detailed statistics. A user can evaluate either the extraction of a set of query interfaces or the matching between a set of query interfaces.

User interfaces were developed to support this step, too. For example, for a given query

interface, VisQI displays the gold standard and the extracted schema trees side-by-side and it highlights the differences between the two.

During evaluation of mappings the system shows the correct matches using a checkbox in the first column of the table. Additionally, the evaluation component adds separate tables of mappings to the user interface that visualize the gold standard mappings and the missed mappings.

The system allows to manage a repository of Deep Web sources. This contains the source files, the schema trees and interface matching information. A user may store these information in files, edit and reload them. She may add the schema tree of a new interface to the gold standard. Furthermore, she can easily update the mapping to reflect the addition of the new interface to the repository.

7 Summary and Outlook

In this final chapter we summarize the thesis. In Section 7.1 we conclude our results of each of the three main contributions. In Section 7.2 we finish the work with an outlook to possible further directions of research in each of the investigated fields.

7.1 Summary

We presented in this work a framework for the integration of Deep Web sources. Deep Web sources offer query interface that are intended for human users. The data of Deep Web sources is hidden behind the query interfaces. Therefore, an automated integration of such Deep Web sources hinges upon the understanding of their query interfaces. Automated interface extraction techniques are the only efficient way to obtain this understanding. Therefore, a main aim of Deep Web integration systems is the extraction of their query interfaces. In Chapter 3 this work provides a novel approach for Deep Web query interface extraction that results hierarchical schema trees. Hierarchical schema trees enable a semantically richer representation of query interfaces than previous approaches did. The presented extraction algorithm performs about 6% better on average than other approaches.

A second central aspect of Deep Web integration is the identification of semantic overlappings among sources to be integrated. This step is called interface matching. A matching approach that exploits the semantics of schema trees outperforms those approaches that rely only on a flat interface representation. The hierarchical structure enables matching techniques to incorporate also structural properties of the tree such as ancestor-relationships among the elements. Therefore, we presented in Chapter 4 a novel matching approach for query interfaces and their elements exploiting the hierarchical representation. Our matching method is not restricted only to single fields as previous approaches, but also identifies mappings between groups of elements directly.

An important task of Deep Web integration, especially when integrating on the fly, is the correct domain classification of interfaces. Similarly to the matching, term-based approaches may predict wrong domains when terms on the interfaces are used homonymously. Instead, we presented in Chapter 5 an approach to classify unknown interfaces to their application domain based on their hierarchical representation. Our approach exploits the mappings introduced before to compute similarities between interfaces.

We developed the system *VisQI* (Visual Query Interface Integration System) that implements all three contributions of our work and therefore enables to perform automatic extraction, mapping and classification for a wide range of interfaces. VisQI provides a graphical user interface that visualizes interfaces, interface trees and mappings. The tool allows an automated evaluation of the achieved results utilizing a large gold standard.

7.2 Future Directions

Future directions in the presented field of research may originate from two main sources: On the one hand, we identified algorithmic weaknesses of the current solution that may be covered in future improvements of the presented methods. On the other hand, the Web and its technologies is in a continuous flow of innovations. New technologies enable new application scenarios and change the appearance of Web sites. Although this work represents the Web at the time of research, we believe that our core ideas are generic and therefore they provide a good framework that easily may be adjusted to cope up with future developments.

Currently the Web is evolving such that it becomes more dynamic. Although, the main communication flow still follows the synchronous HTTP request-respond model, there are developments towards a more asynchronous behavior [4]. The introduction of client-side logic shifts more information towards the clients that should be investigated by extraction and matching. This development is driven mainly by the newer technologies JavaScript and Ajax (see, e.g. [68]). For example, Web sites incorporating these technologies enable dynamic query interfaces that change appearance (such as the number of fields shown or the values given) based on user interaction on the fly. These dynamic technologies process data on client side, therefore this data can be exploited by an advanced extraction algorithm. For example, an extraction system should be able to predict dependencies of fields out of these additional client side data by an additional parsing of the JavaScript portions of Web pages. This use case is important for many application scenarios such as focused crawling [7]. Initial experiments showed, that an improvement of these capabilities is possible without re-engineering lots of the previous achievements.

Although our hierarchical model of query interfaces improves the previous flat data models significantly in its capabilities to express query interfaces semantics it still has potential for further developments. Currently we use layout information for the extraction, but the model does not reflect different layout flavors. For example, in the model we group aligned query interface elements, but we do not distinguish between horizontal and vertical alignment. A finer layout reflection in the model could improve the consecutive steps of integration such as the matching.

Additionally, the current hierarchical model does not cover all types of fields. Some query interfaces enable the user to alternatively fill in particular fields. For example, in the airline domain, there may be a field for a particular airport code and a distinct field for a city name. The user may choose to fill in only one of them. Another example of field types not covered in the current model are query interfaces that provide multiple fields for a single type of information. An example from the real estate domain are interfaces that allow to fill in multiple city names. We may incorporate the latter two cases into an enhanced hierarchical model by introducing different types of edges, for example `XOR`.

Another potential development direction that focuses on a better usability of Web databases are query interfaces that allow the user to flexibly fill in multiple information types into a single field. The disambiguation of the input is done from the Web database.

A sophisticated extraction system should identify such multi-meaning fields. There are two possible solutions for this problem. On the one hand, if rules for disambiguation are defined on client-side (for example, by employing JavaScript), they could be exploited during extraction. On the other hand, the extraction could tag the field with its multiple meanings. These procedure requires an adjustment of the hierarchical query interface model.

We see multiple options for further developments also for the matching step. An enriched query interface model as described above enables better matching methods. For example, in ambiguous matching cases, the matching algorithm may exploit also the layout information. In the current implementation, we exploit only terms and tree structure information. We do not investigate layout information such as alignment or visual distances of the interface elements mapped.

The accuracy of the matching may be improved by including a cross validation based on the tree structures. For example, assume all the children of a single internal node of one interface map to children of a single internal node of a second interface. In this case, it is likely that also the internal nodes map. Moreover, in a group of mappings, if only one mapping conflicts some criteria, it is likely that this mapping is wrong.

Finally, the classification step may be improved when considering more complex patterns that include also other information such as the number of children of a concept set in a pattern. Also, patterns may be refined incorporating layout information such as alignment, visual distances or different font styles.

Both presented domain classification algorithms currently use a relatively simple predication model that may produce ambiguous cases. The pattern classifier relies on the a single best matching pattern whereas the neighbor classifier currently uses only the most similar interface to compare with. Here, an incorporation of the top-k (k>1) nearest patterns or neighbors, respectively, could reduce misclassification and would prevent ambiguous cases when multiple options obey a similar ratio.

Although our efforts represent key issues of Deep Web integration a comprehensive system in that area would need to extend our modules with other components in order to become a standalone end-user application. There are multiple options on how to enhance the current system. When focusing on the establishment of an integrated query interface for human users, merging and labeling components [26, 25] may be plugged in. In contrast, when focusing on a federated information system that enables flexible database-like query interfaces, the results of the matching may be used in a mediator component as implemented in the MiWeb project [15]. Finally, an integration into a Deep Web crawling system is possible.

Bibliography

[1] David Aumueller, Hong Hai Do, Sabine Massmann, and Erhard Rahm. Schema and ontology matching with COMA++. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 906–908, 2005.

[2] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[3] Luciano Barbosa and Juliana Freire. An adaptive crawler for locating hidden Web entry points. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2007.

[4] Luciano Barbosa and Juliana Freire. Combining classifiers to identify online databases. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2007.

[5] Luciano Barbosa, Juliana Freire, and Altigran Soares da Silva. Organizing Hidden-Web databases by clustering visible Web documents. In *Proceedings of the international conference on Data Engineering (ICDE)*, 2007.

[6] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Compututing Surveys*, 18(4):323–364, 1986.

[7] Andre Bergholz and Boris Chidlovskii. Crawling for domain-specific Hidden Web resources. In *Proceedings of the International Conference on Web Information Systems Engineering (WISE)*, 2003.

[8] Michael K. Bergman. The Deep Web: Surfacing hidden value. *The Journal of Electronic Publishing*, 7(1), 2001.

[9] Tim Berners-Lee. Information management: A proposal. Technical report, European Organization for Nuclear Research (CERN), 1989.

[10] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientifc American*, 1(284):34–43, 2001.

[11] Alexander Bilke and Felix Naumann. Schema matching using duplicates. In *Proceedings of the international conference on Data Engineering (ICDE)*, 2005.

[12] J.A. Bondy and U.S.R Murty. *Graph Theory*. Springer Publishing Company, Incorporated, 2008.

[13] Rainer E. Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment problems.* SIAM : Society for Industrial and Applied Mathematics, 2009.

[14] S. Busse, R.-D. Kutsche, U. Leser, and H. Weber. Federated information systems: Concepts, terminology and architectures. Forschungsberichte des Fachbereichs Informatik Nr. 99-9, Technische Universität Berlin, 1999.

[15] Susanne Busse, Thomas Kabisch, and Ralf Petzschmann. MiWeb: Mediatorbased integration of Web sources. Technical report, University of Technology Berlin, 2005.

[16] Michael J. Cafarella, Edward Chang, Andrew Fikes, Alon Y. Halevy, Wilson C. Hsieh, Alberto Lerner, Jayant Madhavan, and S. Muthukrishnan. Data management projects at google. *SIGMOD Records*, 37(1), 2008.

[17] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers. Towards heterogeneous multimedia information systems: the garlic approach. In *Proceedings of the 5th International Workshop on Research Issues in Data Engineering-Distributed Object Management (RIDE-DOM'95)*, 1995.

[18] Chia-Hui Chang. IEPAD: Information extraction based on pattern discovery. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 681–687, 2001.

[19] Kevin Chen-Chuan Chang, Bin He, Chengkai Li, Mitesh Patel, and Zhen Zhang. Structured databases on the Web: Observations and implications. *SIGMOD Records*, 33(3), 2004.

[20] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2001.

[21] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Roadrunner: Towards automatic data extraction from large Web sites. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2001.

[22] Eduard Dragut. *A Framework for Transparently Accessing Deep Web Content.* PhD thesis, University of Illinois at Chicago, Department of Computer Science, 2010.

[23] Eduard C. Dragut, Fang Fang, A. Prasad Sistla, Clement T. Yu, and Weiyi Meng. Stop word and related problems in Web interface integration. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, volume 2, 2009.

[24] Eduard C. Dragut, Thomas Kabisch, Clement Yu, and Ulf Leser. A hierarchical approach to model Web query interfaces for Web source integration. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, volume 2, 2009.

[25] Eduard C. Dragut, Wensheng Wu, A. Prasad Sistla, Clement T. Yu, and Weiyi Meng. Merging source query interfaces on Web databases. In *Proceedings of the international conference on Data Engineering (ICDE)*, 2006.

[26] Eduard C. Dragut, Clement Yu, and Weiyi Meng. Meaningful labeling of integrated query interfaces. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2006.

[27] Dublin Core Metadata Initiative. Dublin core metadata element set, version 1.1. Technical report, DCMI, 2010.

[28] Stefan Endrullis, Andreas Thor, and Erhard Rahm. Evaluation of query generators for entity search engines. *Computing Research Repository (CoRR)*, abs/1003.4418, 2010.

[29] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[30] Christiane Fellbaum. *WordNet: An Electronic Lexical Database (Language, Speech, and Communication).* The MIT Press, 1998.

[31] Daniela Florescu, Alon Y. Levy, and Alberto O. Mendelzon. Database techniques for the World-Wide Web: A survey. *SIGMOD Records*, 27(3):59–74, 1998.

[32] Alon Y. Halevy. Data integration: A status report. In *Proceedings of the Conference Datenbanksysteme in Büro, Technik und Wissenschaft(BTW)*, 2003.

[33] Bin He and Kevin Chen-Chuan Chang. Statistical schema matching across Web query interfaces. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 217–228, 2003.

[34] Bin He, Kevin Chen-Chuan Chang, and Jiawei Han. Discovering complex matchings across Web query interfaces: a correlation mining approach. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, 2004.

[35] Bin He, Tao Tao, and Kevin Chen-Chuan Chang. Organizing structured Web sources by query schemas: a clustering approach. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, 2004.

[36] Hai He, Weiyi Meng, Clement Yu, and Zonghuan Wu. Automatic integration of Web search interfaces with wise-integrator. *The VLDB Journal*, 13(3), 2004.

[37] Hai He, Weiyi Meng, Clement T. Yu, and Zonghuan Wu. Constructing interface schemas for search interfaces of Web databases. In *Proceedings of the International Conference on Web Information Systems Engineering (WISE)*, 2005.

[38] Kevin Chen Chang Bin He and Zhen Zhang. Toward large scale integration: Building a metaquerier over databases on the Web. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2005.

[39] Jun Hong, Zhongtian He, and David A. Bell. An evidential approach to query interface matching on the Deep Web. In *Proceedings of the International Workshop on New Trends in Information Integration (NTII)*, pages 20–23, 2008.

[40] Jun Hong, Zhongtian He, and David A. Bell. An evidential approach to query interface matching on the Deep Web. *Information Systems*, 35(2):140 – 148, 2010. Special Issue: Context-Oriented Information Integration.

[41] Thomas Hornung, Kai Simon, and Georg Lausen. Mashups over the deep web. In *Web Information Systems and Technologies*, volume 18 of *Lecture Notes in Business Information Processing*, pages 228–241. Springer Berlin Heidelberg, 2009.

[42] Matthias Jarke, Y. Vassiliou, P. Vassiliadis, and M. Lenzerini. *Fundamentals of Data Warehouses*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[43] Thomas Kabisch, Eduard Constantin Dragut, Clement T. Yu, and Ulf Leser. Deep Web integration with VisQI. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, volume 3, pages 1613–1616, 2010.

[44] Thomas Kabisch and Mattis Neiling. Wrapping of Web-sources with restricted query interface by query tunneling. In *Workshop on Database Interoperability (InterDB)*, 2005.

[45] Oliver Kaljuvee, Orkut Buyukkokten, Hector Garcia-Molina, and Andreas Paepcke. Efficient Web form entry on PDAs. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2001.

[46] V. Kashyap and A. Sheth. *Information Brokering Across Heterogeneous Digital Data: A Metadata-Based Approach*. Springer USA, 2010.

[47] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[48] Nicholas Kushmerick. Learning to invoke Web forms. In *Proceedings of the joined Conferences CoopIS, DOA, and ODBASE*, 2003.

[49] Laks V. S. Lakshmanan, Fereidoon Sadri, and Subbu N. Subramanian. SchemaSQL: An extension to SQL for multidatabase interoperability. *ACM Transactions on Database Systems*, 26(4):476–519, 2001.

[50] Learning Technology Standards Committee of the IEEE. Draft standard for learning technology - learning object metadata. Technical report, Institute of Electrical and Electronics Engineers, Inc., 2002.

[51] Kristina Lerman, Steven N. Minton, and Craig A. Knoblock. Wrapper maintenance: a machine learning approach. *Journal of Artificial Intelligence Research*, 18(1):149–181, 2003.

[52] Ulf Leser. *Query Planning in Mediator Based Information Systems.* PhD thesis, TU Berlin, Fachbereich Informatik, 2000.

[53] Ulf Leser and Felix Naumann. *Informationsintegration: Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen.* dpunkt-Verlag, Heidelberg, Germany, 2006.

[54] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[55] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1996.

[56] Chen Li, Ramana Yerneni, Vasilis Vassalos, Hector Garcia-Molina, Yannis Papakonstantinou, Jeffrey Ullman, and Murty Valiveti. Capability based mediation in tsimmis. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1998.

[57] W. Litwin, A. Abdellatif, A Zeroual, B. Nicolas, and Ph. Vigier. Msql: a multidatabase language. *Information Science*, 49(1-3), 1989.

[58] Witold Litwin, Leo Mark, and Nick Roussopoulos. Interoperability of multiple autonomous databases. *ACM Comput. Surv.*, 22(3), 1990.

[59] Bing Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data.* Data-Centric Systems and Applications. Springer, 2007.

[60] Bing Liu, Robert Grossman, and Yanhong Zhai. Mining data records in Web pages. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 601–606, 2003.

[61] Bing Liu and Yanhong Zhai. Net - a system for extracting Web data from flat and nested data records. In *Proceedings of the International Conference on Web Information Systems Engineering (WISE)*, pages 487–495, 2005.

[62] Yiyao Lu, Hai He, Qian Peng, Weiyi Meng, and Clement T. Yu. Clustering e-commerce search engines based on their search interface pages using wise-cluster. *Data and Knowledge Engineering*, 59(2):231–246, 2006.

[63] Jayant Madhavan, Shawn R. Jeffery, Shirley Cohen, Xin Dong, David Co, Cong Yu, and Alon Halevy. Web-scale data integration: You can only afford to pay as you go. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2007.

[64] Christopher D. Manning and Hinrich Schütze. *Foundations of statistical natural language processing.* MIT Press, Cambridge, MA, USA, 1999.

[65] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2002.

[66] Xiaofeng Meng, Dongdong Hu, and Chen Li. Schema-guided wrapper maintenance for Web-data extraction. In *Proceedings of the Internatioinal Conference on Web Information and Data Management (WIDM)*, 2003.

[67] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill Higher Education, 1997.

[68] Tom Negrino and Dori Smith. *JavaScript and Ajax for the Web*. Peachpit Press, Berkeley, CA, USA, 2006.

[69] Zaiqing Nie, Ji R. Wen, and Wei Y. Ma. Object-level vertical search. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2007.

[70] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall, 1999.

[71] M. F. Porter. An algorithm for suffix stripping. *Readings in Information Retrieval*, pages 313–316, 1997.

[72] Sriram Raghavan and Hector Garcia-Molina. Crawling the Hidden Web. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2001.

[73] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[74] Juan Raposo, Alberto Pan, Manuel Alvarez, and Justo Hidalgo. Automatically generating labeled examples for Web wrapper maintenance. In *Proceedings of the International Conference on Web Intelligence (WI)*, 2005.

[75] Juan Raposo, Alberto Pan, Manuel Alvarez, and Angel Vina. Automatic wrapper maintenance for semi-structured Web sources using results from previous queries. In *Symposium on Applied Computing (SAC)*, 2005.

[76] Mary Tork Roth and Peter M. Schwarz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1997.

[77] Ingo Schmitt. *Schemaintegration für den Entwurf föderierter Datenbanken*, volume 43 of *DISDBIS*. Infix Verlag, St. Augustin, Germany, 1998.

[78] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, 1990.

[79] Stefano Spaccapietra, Christine Parent, and Yann Dupont. Model independent assertions for integration of heterogeneous schemas. *The VLDB Journal*, 1(1):81–126, 1992.

[80] Andreas Thor, David Aumueller, and Erhard Rahm. Data integration support for mashups. In *Sixth International Workshop on Information Integration on the Web (IIWeb)*, 2007.

[81] World Wide Web Consortium (W3C). W3C Semantic Web Activity. http://www.w3.org/2001/12/semweb-fin/w3csw, 2001.

[82] Jiying Wang, Ji-Rong Wen, Fred Lochovsky, and Wei-Ying Ma. Instance-based schema matching for Web databases by domain-specific query probing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

[83] Gio Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3), 1992.

[84] World Wide Web Consortium (W3C). Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl/, 2001.

[85] World Wide Web Consortium (W3C). W3C Web Services Activity. http://www.w3.org/2002/ws/, 2002.

[86] World Wide Web Consortium (W3C). XHTML 1.0: The Extensible HyperText Markup Language (Second Edition). http://www.w3.org/TR/html/, 2002.

[87] World Wide Web Consortium (W3C). Resource Description Framework (RDF): Concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts/, 2004.

[88] World Wide Web Consortium (W3C). Recommendations of Simple Object Access Protocol. http://www.w3.org/TR/soap/, 2007.

[89] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0 (Fifth Edition). http://www.w3.org/TR/xml/, 2008.

[90] World Wide Web Consortium (W3C). OWL2 Web Ontology Language. http://www.w3.org/TR/ow2-overview/, 2009.

[91] Wensheng Wu, Clement Yu, AnHai Doan, and Weiyi Meng. An interactive clustering-based approach to integrating source query interfaces on the Deep Web. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[92] Yanhong Zhai and Bing Liu. Web data extraction based on partial tree alignment. In *Proceedings of the International Conference on World Wide Web (WWW)*, pages 76–85, New York, NY, USA, 2005. ACM.

[93] Zhen Zhang, Bin He, and Kevin Chen-Chuan Chang. Understanding Web query interfaces: best-effort parsing with hidden syntax. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

Selbständigkeitserklärung

Ich erkläre hiermit, dass

- ich die vorliegende Dissertationsschrift "Extraction and Integration of Web Query Interfaces" selbständig und ohne unerlaubte Hilfe angefertigt habe,
- ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe oder einen solchen besitze, und
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin gemäß Amtl. Mitteilungsblatt Nr. 34/2006 bekannt ist.

Berlin, den 24.01.2011